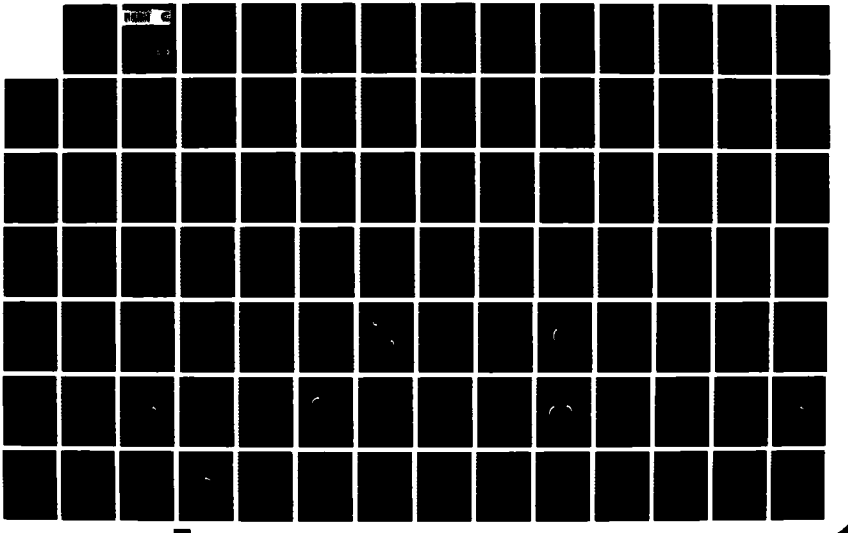
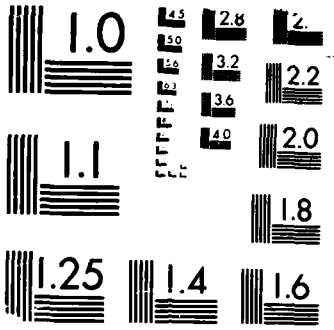
AD-A193 796 HIGHER ORDER LANGUAGES FOR ROBOTS(U) MANUFACTURING TECHNOLOGY INNORMATION ANALYSIS CENTER CHICAGO IL J R BLAHA ET AL. OCT 86 MTIAC-SOAR-86-01 1/2

UNCLASSIFIED DLA900-84-C-1508 F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963-A

(2)

MTIAC

MTIAC-SOAR-86-01

HIGHER ORDER LANGUAGES FOR ROBOTS

October 1986

James R. Blaha
John P. Lamoureux
Keith E. McKee, Ph.D.
IIT Research Institute

Prepared By:
Cresap, McCormick and Paget
Division of TPF&C, Inc.
(312) 567-4730
Under Contract to
Department of Defense
Contract Number DLA900-84-C-1508

A Department of Defense Information Analysis Center

Overview

MTIAC is a Department of Defense (DoD) Information Analysis Center. MTIAC serves as a central source for currently available and readily usable data and information concerning manufacturing technology. The primary focus of the Center is to collect, analyze, and disseminate manufacturing technology for the production of defense materials and systems.

The funding agency for MTIAC is the Defense Technical Information Center of the Defense Logistics Agency of the Department of Defense, in Alexandria, Virginia. MTIAC's data collection and dissemination function is tied to DTIC by a shared bibliographic data base.

The DoD supports manufacturing technology programs conducted by the Air Force, Navy, and Army as well as by the Defense Logistics Agency. MTIAC's role is to support the effective use of manufacturing technology by DoD agencies and the industrial contractor base, at both the prime contract and subcontract level. This support is provided through a range of services from technical inquiries to bibliographic searches and special tasks within the scope of the contract. Services are offered on a fee-for-service basis to subscribers and nonsubscribers.

MTIAC Objectives

The Department of Defense established the Manufacturing Technology Information Analysis Center (MTIAC) through the Defense Logistics Agency to improve productivity, reduce costs, and reduce lead times in the production of defense equipment and to further the use and development of advanced technologies. By consolidating and retaining manufacturing information and experience in a central repository staffed by manufacturing specialists, knowledge can be disseminated and applied quickly and effectively to plant modernization programs. The Center benefits engineers and information specialists, government agencies, and defense contractors by saving valuable man-hours in locating data and information and applying the new technologies. The result can be reduced planning and/or production costs.

MTIAC also serves the civil sector within the constraints of the priorities of defense needs and limits on disseminating information, because of security classification, and the export laws and regulations on technology transfer.

Scope of the Program

Activities Scope

MTIAC performs these activities:

- Maintains a bibliographic data base on manufacturing technology
- Maintains a DoD Manufacturing Technology Program (MTP) data base
- Prepares and publishes handbooks, data books, reference works, state-of-the-art reviews (SOARs), critical reviews and technology assessments, conference proceedings, newsletters, and other publications
- Responds to technical, bibliographic, and other user inquiries
- Establishes and maintains programs of awareness and visibility of MTIAC capabilities and services to promote the Center's use
- Performs special tasks for government users, separately funded through the MTIAC contract.

Further information regarding MTIAC services, products, subscription plan or additional copies of this report may be obtained by writing or calling: MTIAC, IIT Research Institute, 10 West 35th St., Chicago, IL 60616 (312) 567-4730

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release. Distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) SOAR-86-01	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Cresap, McCormick and Paget Division of TPF&C Inc.	6b OFFICE SYMBOL *(If applicable)*
6c ADDRESS (*City, State, and ZIP Code*) 10 West 35th Street Chicago, IL 60616	7a NAME OF MONITORING ORGANIZATION Dr. Lloyd Lehn Acquisitions & Logistics PSIR
	7b ADDRESS (*City, State, and ZIP Code*) Off. of the Asst. Secy. of Defense Pentagon, Room 3C257 Washington, DC 20302-8000
8a NAME OF FUNDING/SPONSORING ORGANIZATION Defense Logistics Agency	8b OFFICE SYMBOL *(If applicable)*
	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DLA-900-84-C-1508
8c ADDRESS (*City, State, and ZIP Code*) Cameron Station Alexandria, VA 22304-6145	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO. / WORK UNIT ACCESSION NO

11 TITLE (*Include Security Classification*)
Higher Order Languages for Robots

12 PERSONAL AUTHOR(S)
K. E. McKee, J. P. Lamoureux, J. R. Blaha

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (*Year, Month, Day*)	15 PAGE COUNT
SOAR	FROM _____ TO _____	October 1986	137

16 SUPPLEMENTARY NOTATION
Hardcopy available from MTIAC only. Reproduction not authorized except by permission.

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	02	
13	08	

18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
Robots
Manufacturing
Higher Order Languages
Robot Programming

19 ABSTRACT (*Continue on reverse if necessary and identify by block number*)

This state of the art review of higher order languages for robots provides a background of robot programming approaches. Strengths and weaknesses of various approaches are presented. The second section of the report discusses specific languages. Manipulator level languages developed by universities and research institutes include: AL, PAL, JARS, LAMA-S, LM, ROBEX, RPL, VML, LENNY, LPR, MAL, RCCL, RCL, SRL, and LMAC. Commercial languages are: VAL, ML, EMILY, SIGLA, AML, HELP, MCL, AML/E, AML/V, RAIL, PASRO, CIMPLER, VAL II, AR-BASIC, KAREL and AML/X. Task level languages discussions include RAPT and AUTOPASS. CAD/GRAPHIC systems that are discussed include GRASP, IGRIP, GRIPS from Universities and PLACE, ANIMATE COMMAND, ADJUST BUILD and ROBOT-SIM from commercial manufacturers. A third section discusses robot performance and the elements of a robot programming system. The resulting programming requirements are set forth. Comparisons are drawn for VAL, VAL-II, RAIL, KAREL, and AML/X. Off-line programming is compared to teach programming. DoD HOL issues and ongoing research are discussed. Conclusions emphasized the needs for the language and the requirements of the entire robot system.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Thomas B. Turner	22b TELEPHONE (*Include Area Code*) (312)567-4730 / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

PREFACE

This state-of-the art review was prepared by the Manufacturing Technology Information Analysis Center (MTIAC) under Contract DLA-900-84-C-1508 for the Department of Defense.

Higher order languages are defined as textual languages in this study. However, the study also highlights the trend toward the use of teach pendants which have incorporated in them some higher order capabilities, noting that the choice depends upon organizing perception of user friendliness. A major portion of the review is dedicated to a discussion of currently available robot programming languages. The efforts of universities and research institutes are reviewed as are commercial offerings. Languages are discussed in terms of levels (servo, manipulator, and task). CAD/Graphic systems are also covered.

The discussion of languages leads to a section on language comparison. Language comparison constitutes the principal portion of the study and sets forth the basis for comparison, a listing of elements to be compared, and a comparison of several robot programming languages (VAL, VAL II, RAIL, KAREL, and AML/X).

Completion of the language comparisons permitted this review to summarize some of the current issues in the field. Issues include the relationship to teach vs off-line programming, language implementation, trade-offs necessary as languages approach task level, and the appropriate scope of the robot programming languages when considering manufacturing cells as opposed to stand-alone installations.

The state-of-the-art review concludes with a discussion of special robot programming considerations associated with defense manufacturing. Batch manufacturing, communication between contractors and subcontractor, documentation of such information as cell status, and the impact of remanufacturing are cited as relevant.

Recommendations suggest research in areas such as the need for a programming methodology that recognizes the totality of robot programming requirements, world modeling, simulation, communication, sensor technology and safety. The conclusions emphasize the importance of supporting the entire robot programming system, noting that if a program is to be written directly in a language, the language needs to have both high-level constructs for ease of programming and low-level constructs for explicit control. In the same sense, programs written using an applications interface need a language that supports a flexible user interface.

The content of this review has been derived from a review of over 50 references. However the subject of robot programming is constantly evolving so that the literature had to be supplemented by 13 field interviews with specialists in robot programming and languages. The interviews are listed in an appendix. Upon completion, the document was reviewed by experts in the field. The writers particularly wish to thank Dr. Lloyd Lehn, Office of the Assistant Secretary of Defense, Acquisition and Logistics, for his guidance and review. Additionally the authors appreciate comments from the following reviewers:

Dr. Graham H. Morris
National Bureau of Standards

Dennis C. Haley
Martin Marietta

Bertil Thorvaldsson
ASEA Robotics Inc.

Dr. Margaret A. Eastwood
CIMCORP

Dr. Robert L. Haar
General Motors

Mitchell Ward
GM Fanuc Robotics

This state-of-the-art review is one of a series of reports intended to keep MTIAC users abreast of technology and information within the scope of Manufacturing Technology. Information about other reports may be obtained by contacting:

MTIAC
c/o Cresap, McCormick and Paget
Division of TPF&C Inc.
10 West 35th Street
Chicago, IL 60616

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

LIST OF FIGURES

LIST OF TABLES

1. BACKGROUND

1.1 INTRODUCTION

The purpose of this paper is to present a state-of-the-art review of higher order languages for robots. Research was conducted by a study of published literature and by interviews with industry professionals. The literature consisted of papers published in technical journals and presented at conferences, industrial handbooks, and textbooks. Interviews were conducted with professionals from the defense community (primarily the services and aerospace contractors), the nondefense community (primarily automotive industry), universities, and the robot vendors.

The subject of robot programming languages is not mature. Consequently, there are conflicting opinions on various issues and different approaches have been explored. This creates some confusion, particularly for nonprogrammers, when trying to evaluate the strengths and weaknesses of various languages. The intent of this review is to illustrate the current issues in robot programming, identify the languages, and provide a method by which languages can be evaluated.

The review is structured into six sections. Section 1 provides a background of robot programming approaches, including the strengths and weaknesses of different techniques. Section 2 identifies and describes robot languages that are commercially available and those that have been developed by universities and research institutes. Section 3 presents a method by which robot languages may be evaluated. In addition, some comparisons that have been published in the literature are examined. Section 4 discusses some issues relevant to defense manufacturing. Recent and ongoing research is discussed in Section 5. Finally, conclusions and recommendations are presented in Section 6.

1.2 SCOPE

The objective of this paper is to review the state of the art in higher order programming languages for robots. The scope is limited to languages that apply to industrial robots in manufacturing applications. Different types of robots will have different programming language requirements. A

manufacturing robot, for example, operates in a very structured and predictable environment when compared with that of a battlefield robot. Although the specific language requirements are different, the general issues discussed in this paper are generally applicable to all types of robots.

There is by no means an industrywide definition for "higher order language" for robots. Definitions range from an "off-line programming language" to "any language containing higher level constructs" to "task level language." The definition providing the clearest distinction of what constitutes a higher order language is that which distinguishes between teach pendant programming and off-line textual languages. By definition then, all higher order languages are characterized as textual and man-readable, though the environment in which they are generated may be off- or on-line with the robot controller. Teach pendant programming is discussed, however, to illustrate the evolution and benefits of higher order languages and to fill in the backdrop for the current debates over robot programming techniques.

1.3 COMPUTER PROGRAMMING LANGUAGES

One school of thought promotes the use of general purpose computer programming languages as the base for robot programming languages. In fact, the evolution of robot programming languages closely parallels that of computer languages. A review of the evolution of programming languages is therefore helpful in providing a perspective from which to evaluate current and future robot languages.

The earliest programming, using a machine-specific language, required intimate and detailed knowledge of the computer hardware. Machine language programming involved coding instructions in binary format (a series of 1s and 0s), and each machine had its own format for particular instructions. Editing was often done through a series of toggle switches on the front panel of the computer. There were no high level instructions, so even a simple operation like adding two numbers involved a series of instructions specifying where the data was to be retrieved from, what was to be done, and where the result was to be stored. A major portion of a programmer's time and energy was spent on mentally translating his program concept, such as "addition," into the corresponding binary codes that would literally activate the computer circuitry necessary to access, manipulate, and store data.

To relieve the programmer of some of the details of machine programming, a program called an "assembler" was developed. This first programming tool allowed the creation of programs in a more symbolic fashion using mnemonics for operation codes and names for data locations. The assembly language program would be automatically translated by the assembler into the proper binary machine codes for input to the target execution computer.

Assembly programming is still widely practiced today, for two major reasons. An assembly program has a nearly one-to-one relationship between its instructions and the translated machine language instructions. Therefore, assembly programs yield executable programs that are as space and time efficient as programs coded directly in machine code.

The other major reason is that hardware attached to computers, such as printers, terminals or servo drives, requires very low-level machine-specific control signals for operation. The specific requirements of control demand that a program be capable of manipulating memory locations very explicitly, just as machine code programs do. Consequently, assembly programming has a place wherever the most efficient and the most hardware-oriented programming is required. Many of the motion control algorithms in robotics are written in assembly code because speed of execution is essential.

A major evolutionary step in programming languages came in 1953, when IBM wrote the first "compiler" for a language called FORTRAN. This first high-level language was designed to perform "Formula Translation" for engineers; that is, it converted a series of calculations into a program for solution by a computer. A FORTRAN compiler, like any compiler, translates a program written in a higher-level, problem-oriented symbolic language into a lower-level, hardware-oriented language such as assembly. During the translation process, other programs or data may be brought together or compiled in a single output file called an object program.

Thousands of such high-level languages have been developed over the years, sometimes in conjunction with the corresponding compiler/translator, as was the case with FORTRAN, or sometimes as a stand-alone item requiring major independent initiatives to develop a compiler, as with the Ada language. The reason a language is different than its implementation (the compiler) is that the language is problem-oriented while the implementation must necessarily be hardware-oriented. This division allows for many languages to be usable on a

variety of computers by virtue of forcing compilers to address the hardware implementation issues, leaving the high-level programming language free to express just the problem itself.

The left side of Figure 1 shows how the simple problem "2 + 3 = ?" is programmed in the high-level language FORTRAN, and what the resulting assembly code is after processing by a FORTRAN compiler for a VAX 11/785 computer. The last stage shows what the actual machine language program looks like after processing by an assembler. The point of detailing out these processing stages is that a programmer may have programmed the solution to the question "2 + 3 = ?" in either the FORTRAN, assembly, or machine languages directly. An even higher level of programming is also illustrated, as a goal driven query processor. In the earliest days of programming, the only computer language available was machine, but 30 years later thousands of languages can express the same problem in a more or less concise manner as the given FORTRAN program.

The high-level general purpose computer languages have evolved over the years, and robot programming languages have paralleled this evolution. Unstructured languages such as BASIC and FORTRAN were the first to be developed. These were followed by the structured languages such as PASCAL and C. Many present day robot languages are based on the concept of structured languages. Currently, functional languages and object-oriented languages are being developed. Gini and Gini (1)* suggested that future robot languages will be based on the concepts of these languages. This is due, in part, to an anticipated change in the traditional Von Neumann computer architecture which will occur in the near future.

High-level languages are independent of the computer hardware. A program written in a high-level language will run on any computer with the appropriate translator. There are two methods of translation. An interpreter translates one instruction, executes it, then translates the next instruction, executes it, and so on. The advantage is that the programmer can edit and then execute a few instructions at a time, which makes debugging programs easier.

*Numbers in parentheses refer to list of references at the end of this report.

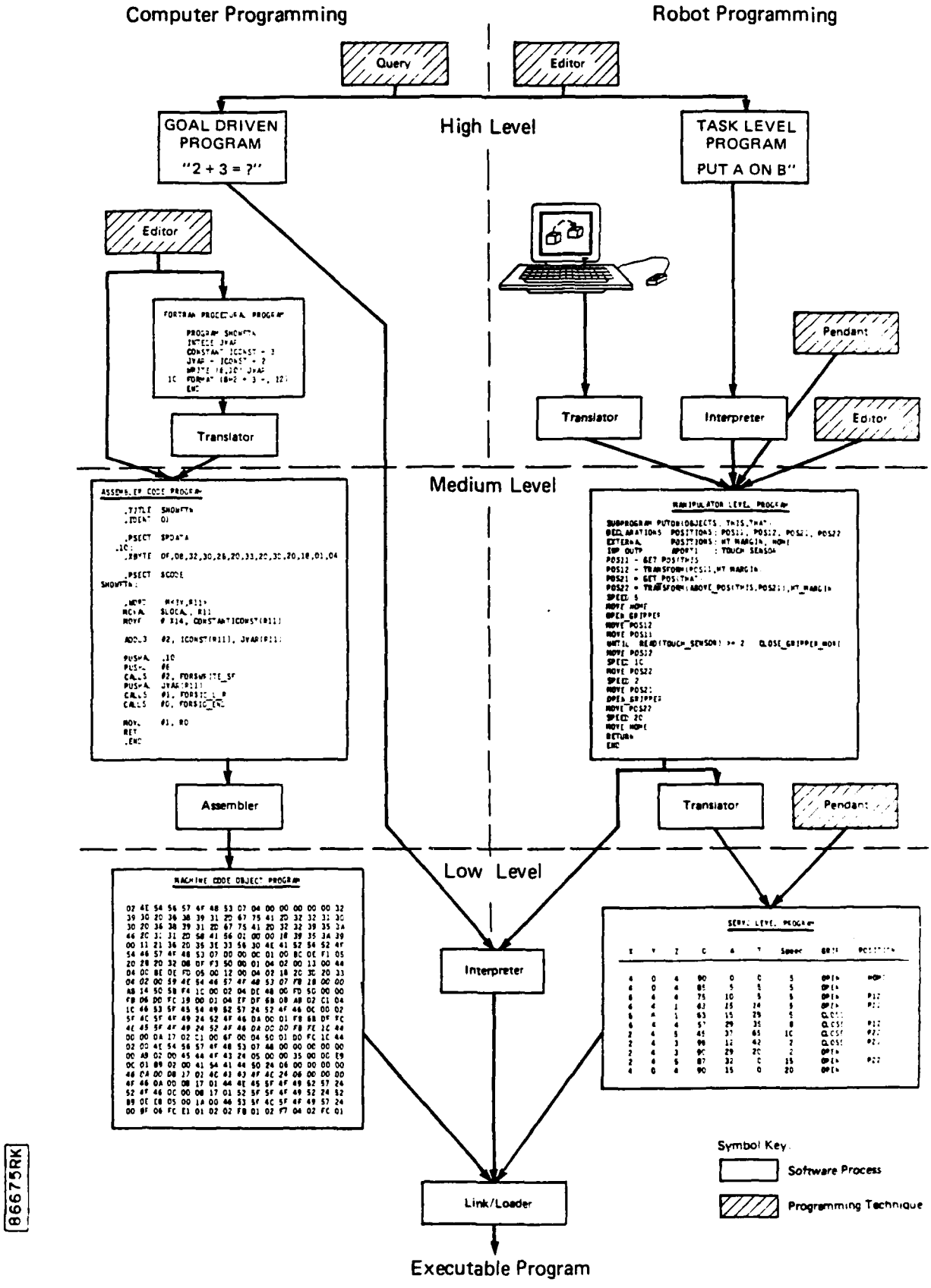


Figure 1. Comparison of computer and robot programming levels and techniques. (Source: IITRI).

The disadvantage is that interpreted programs run more slowly and are less memory efficient. A compiler, on the other hand, reads the entire program, translates it, and produces an object code, which is an executable machine level program. The advantage to a compiler is that it produces a program that executes faster and is memory efficient. The disadvantage is that debugging is a slow process because the entire program must be recompiled each time it is edited. BASIC is an interpreted language and FORTRAN is a compiled language. Those familiar with these languages can appreciate the tradeoffs between ease of program development and speed of execution. The issue of interpreted versus compiled languages is very relevant to robotics, because some languages are interpreted and others are compiled.

The next level up is that of the application programs. These are computer programs, usually written in a high-level language, that provide a user-friendly interface for performing some task. Spreadsheets and word processors, for example, are applications programs. They allow someone unfamiliar with computer programming to perform operations on a computer. The actual computer program being executed is transparent to the user. In much the same way, there are interactive robot programming packages which run on graphics systems and computer aided design (CAD) systems. The actual robot program generated by these packages is transparent to the robot programmer. This topic is explored in more detail in Section 1.6.4.

The most generic criteria for evaluation of computer languages include the ease of programmability and the degree to which the language is open to the environment. More specifically, Pratt (2) discusses the following eight issues generic to all programming, which will later be shown as relevant to robot programming:

1. Clarity and simplicity of the language concepts
2. Clarity of syntax
3. Naturalness for the application
4. Support for abstraction
5. Ease of testing/verification/simulation
6. Programming environment features related to the language
7. Portability
8. Cost of development, execution, maintenance

Every language designer considers these criteria to a greater or lesser degree. Similarly, a programmer examines the range of available languages with the same criteria in mind, as each new application arises. Given the number of criteria here, and the subjectivity of them, it is no wonder that there are so many languages and that the effort involved in picking one for a particular application is formidable. The same is true of robot programming languages.

If we examine robot programming with these criteria in mind, it becomes apparent that a few of these have been thoroughly addressed, while others have been virtually ignored. Conventional data processing languages also tend to selectively address these points, but it must be remembered that those languages have a substantial history of design, development, and revision. Relative to robot languages, conventional languages are much more mature and standards do exist, i.e., FORTRAN 77, COBOL 66, and Ada.

As a final note, it is important to realize that there are two distinct environments in computer programming: the programming environment and the operating environment. The programming environment consists of the hardware, software, and tools used to develop a program. The operating environment consists of the hardware and software involved in the execution of the program. To clarify the point, consider a commercial spreadsheet. The programming environment is located at the vendor's site, and consists of the computer and tools used in developing the product. The operating environment, on the other hand, is at the customer's site, and consists of the computer, operating system programs, and equipment on which the product is used. The programming and operating environments are impacted by the implementation of the language. As noted earlier, an interpreted language is easier to debug (programming environment) but slower in execution (operating environment) than a compiled language. In robotics, the programming and operating environments are extremely important considerations in the language. This issue is explored further in Sections 3.1 and 3.2.

1.4 NUMERICAL CONTROL LANGUAGES

Another school of thought has promoted the use of APT-like languages for use in robot programming. The basic foundations of this argument are that robots are similar to NC machine tools, and there is a large base of experienced APT programmers in industry.

NC languages, such as APT, were developed specifically for control of machine tools. In effect, APT-like languages are high-level computer languages that allow the programmer to specify the machining operations in a generalized fashion applicable to a class of machine tools. A programmer writing an APT program is not concerned with the particular brand of machine tool controller on which the program will run, but only with specifying a sequence of functions to be performed within the functional constraints of the tool. The functions performed are primarily concerned with motion control and tool statements. A postprocessor translates the APT program into a set of machine executable statements in much the same way that a compiler translates a standard FORTRAN program into machine code executable by a specific computer. The postprocessor must take into account details of a particular machine tool controller combination. This requires that each machine have an APT postprocessor, and that it reside somewhere other than in the machine tool controller itself, because the controller is incapable of running such translation programs. This has led to a problem where a user must maintain a large library of postprocessors on some computer such as a CAD system. A solution to this problem has been the introduction of the RS494 standard for cutter location (CL data). Essentially, CL data is a standardized output of an APT postprocessor which is input to a machine tool controller. The controller then takes care of any machine-specific translations internally (see Figure 2). This is worth noting, since a similar standard for robot programs called IRDATA has been proposed in Germany (3) (4).

1.5 ROBOT PROGRAMMING LEVELS

Just as there are various levels of computer programming, there are various levels of robot programming. The very nature of programming demands that problems be representable in a language natural to the level of interface with the man or machine. Thus, a subset of the English language is a good choice for application programming, such as the trivial query "2 + 3 = ?" in

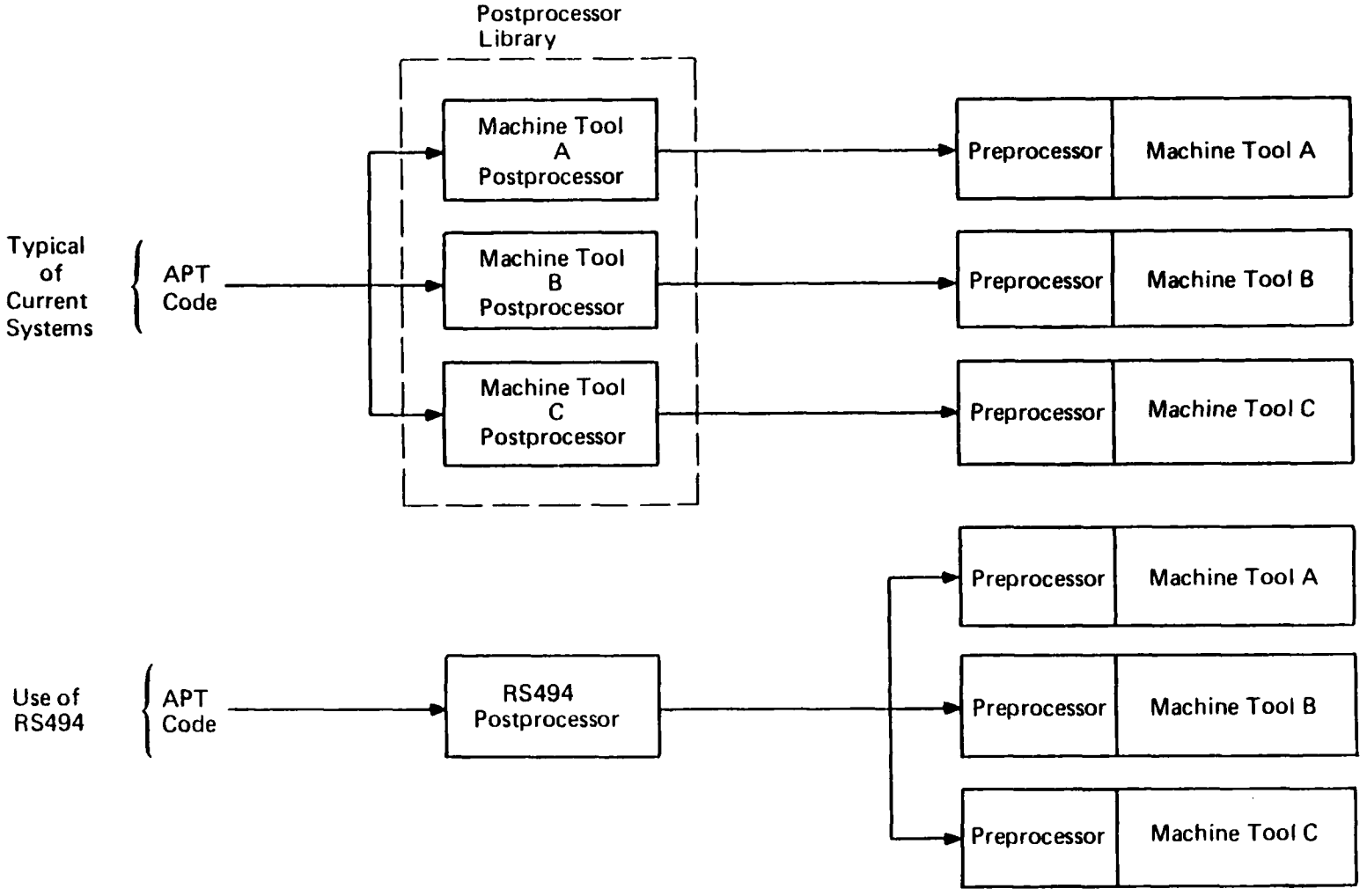


Figure 2. Use of the RS 494 Standard. (Source: IITRI).

Figure 1, while rigidly defined binary codes are more suitable to direct control of binary digital computer states. Although it is desirable for the user to express the problem in a high-level textual fashion, it is impractical for a digital computer to interpret such an expression directly. The conversion process, called translation, is often used off-line to decompose high-level user programs into correspondingly lower levels of commands that can be easily and unambiguously executed by the computer or controller. This process does not burden the runtime system, because the translation process occurs off-line. Another method of runtime execution, called interpretation, performs a similar translation process immediately before the execution of individual program statements. Interpretation may significantly affect the overall program execution speed, though improvements in hardware design indicate that this potential bottleneck will all but vanish in the near future.

In the computer programming domain, a user program normally goes through two translation stages before becoming "executable machine code." Similarly, in the robotics arena, translation is the more frequently used process than interpretation. The left side of Figure 1 shows what the typical three levels of computer programming languages look like. Each of the three programs is semantically identical. A programmer might have written his solution to the problem "2 + 3 = ?" in either of the three levels, or submitted it to the goal driven query processor. It should be obvious which level is most desirable. The remainder of this section explains how robot programming may also occur at various levels, with a variety of techniques, as indicated by the right side of Figure 1.

Bonner and Shin (5) describe five levels of robot programming: task-oriented, structured programming, primitive motion, point-to-point, and microcomputer. The microcomputer level is the lowest, and consists of servo commands and sensor interfaces. Point-to-point is a program consisting of commands similar to those generated in some teach pendant programming. This consists of endpoints for motion and, possibly, specification of the path between the endpoints. Primitive motion programming is expanded beyond that to include simple branching, subroutines, and primitive parallel execution. Structured programming is the level containing constructs similar to those found in structured programming languages. A program at this level is still in terms of motions and end-effector commands. The highest level is the task

level. At this level the program exists in terms of tasks to be performed rather than motion sequences. Motion is defined in terms of coordinate frame transformations, parallel execution of tasks is possible, and the concept of state variables which describe the environment exists. Within this framework, most current programming languages reside at the primitive motion and structured programming levels.

Yong, Gleave, Green, and Bonney (6) consider four levels: joint, manipulator, object, and objective. They actually consider one level higher than Bonner and Shin by proposing the objective level. At this level a task is specified at a very general level such as "spray interior of car door." This is a goal that is broken down into a series of tasks. Within their framework, most current programming languages reside at the manipulator level.

Other authors use different levels, but all of the approaches are very similar. The important thing is to note that there are basically three levels in robot programming that represent distinct conceptual differences in program content. We term these the servo level, the manipulator level, and the task level.

At the lowest level, the servo level, a program consists of a series of endpoints, speeds, and input/output commands. Each endpoint is represented as a group of joint coordinates, so that a six-axis robot has six data values representing one position. This type of program, which is often produced by a teach pendant, is robot specific. Input/output commands typically entail reading a switch closure (input) or energizing a relay (output) after a motion has been executed. The path between endpoints is generated by the robot controller by calculating a series of intermediate points between the endpoints. The servo control then causes each joint to drive to its target position at the first intermediate point. When all joints have achieved their target position, the robot drives to the next intermediate point, and so on, until the final endpoint is reached. In terms of robot languages, it is important to remember is that the program must specify a series of individual joint positions. While this level of programming is quite adequate for some applications, trying to program an assembly task at this level would be virtually impossible.

At the next level, the manipulator level, a program has explicit motion commands required to perform a task, some sensor capability, and branching and loop constructs for error handling. It is at this level that most programming languages reside today. Motion is programmed using commands that specify moving from point A to point B, where points A and B might be specified by Cartesian coordinates. The joint positions required to achieve this motion are calculated when this level is decomposed (translated) into the servo level commands, and this is transparent to the programmer. Variables, macros, procedures, and looping constructs allow the programmer to write a small section of code to do repetitive tasks such as palletizing. This is in contrast to the servo level, where each position in the pallet must be explicitly stated in the program. Conditional statements like "if sensor = 0 go to DROP-PART," allow for branching control of the program to different sections of code based on measurement of some input, an essential characteristic for handling errors. The manipulator programming level is much more powerful than the servo level, but does have some disadvantages. These are explored in more detail in Section 1.6.2. The important thing to remember at this point is that in order to perform some function, the programmer must explicitly specify all motions and gripper actions.

At the highest level, the task level, a program is a task specification. At this level the programmer may specify "put box A on box B" and decomposition will generate a list of all motions required to perform the task. The output of the decomposition is a manipulator level program. The key distinction between task and manipulator levels is that task level programs are stated in terms of the objects to be manipulated and the tasks to perform, while manipulator level programs are stated in terms of the motions required to achieve that task. Some current robot languages have limited task level capabilities, but no task level language exists. Many problems need to be solved before task level programming becomes a reality, and these are explored in detail in Section 5.3.

1.6 ROBOT PROGRAMMING TECHNIQUES

A great source of confusion in the literature is the failure to distinguish between robot programming levels and robot programming techniques.

Figure 1 compares the levels at which computer programs are created and the corresponding levels of robot programs. It also shows the programming techniques used to create computer programs versus robot programs. Programming techniques consist of on-line teach pendant programming, off-line textual programming with an editor, and off-line programming using a graphics or CAD system. Each programming technique results in a program that resides at some level. Also, it is important to note that a certain technique can result in programs at different levels. For example, some teach pendants generate manipulator level programs, while other (older) pendants generate servo level programs. This is analogous to writing a FORTRAN program and an assembly program with an editor. The same technique is used to write both programs, but they are at different levels. This section summarizes current robot programming techniques, their relative advantages and disadvantages, and how they relate to the three levels discussed in Section 1.5.

1.6.1 Teach Programming

One of the oldest programming methods for robots is the teach mode. This was developed before general purpose computers were available for robot controllers. Teaching is usually done by an operator on the shop floor, and can be done by literally moving the robot gripper to achieve the desired pattern. For example, to program a spray painting robot the programmer (usually a shop floor operator) leads the robot through all of the motions for locating, moving, and orienting the spray gun. Two types of path control are possible with this programming method: continuous path and point-to-point. Point-to-point can be further broken down into controlled path and coordinated motion. In continuous path control, a series of closely spaced points is recorded as the programmer moves the robot, and the robot will repeat the prescribed pattern with all of the details and idiosyncrasies present in the programmer's motion. With controlled path, the programmer specifies the endpoints and the type of path (straight line or sometimes circular) to follow between the endpoints. With coordinated motion the path is unspecified, but all axes arrive at their endpoints simultaneously.

Teach programming can also be accomplished with a teach pendant, a hand-held device enabling the operator to position and orient the robot by watching, but not directly touching, the robot. Motion is controlled with a series of push-buttons on the pendant. Either each axis is controlled separately or, in some cases, the robot can be moved in a straight line in rectangular, cylindrical, or spherical coordinate space. When the robot has been positioned at the desired location, the endpoint is recorded by the programmer.

There are advantages and disadvantages to teach programming. The primary advantage is that it is easy to learn. The programmer can be an operator on the shop floor who is familiar with the application, as opposed to a trained computer programmer. Also, it is relatively easy to program a path in a complex geometric situation with many obstacles. Since the programmer can position the robot under load, repeatability is the only source of error when the program is executed. There are also several disadvantages to teach programming. The robot is rendered unproductive during programming since the robot itself is being used. Programming cannot rely on previously developed general purpose procedures to perform tasks, such as those provided by subroutine libraries. Modular program development is hampered for similar reasons as noted in Section 3.2. Programming can be dangerous since the operator is close to the robot when teaching and debugging the program. Finally, teach programming will be unsuitable for use in a highly automated and flexible environment, such as a small batch automated factory of the future.

Another technique of robot programming that should be mentioned here is off-line teach programming. This involves using a duplicate robot in a replica of the manufacturing environment. The program is generated using the teach mode in this simulated environment while the production robot continues to work. Later, the program is transferred to the shop floor and the positions are "fine tuned" to account for differences between the actual shop floor and the mock-up. In a production operation using many similar robots in a relatively uncomplicated environment, this can be a viable programming system.

The program generated by some of the older teach pendants is at the servo level. The program is a set of joint coordinates, speeds, and activation signals. It is not very man-readable, which is the reason editing is done in conjunction with the robot. Typically, the robot controller program will read this servo level program and generate the appropriate signals to activate the

servo drives. With recent improvements in hardware leading to increased processing speeds, some vendors are electing to write high-level language code to interpret the servo level program.

Teach pendants of recent years are considerably more powerful. Some are capable of generating and editing manipulator level programs, teaching points as needed, and controlling virtually every element in the workcell. The technique of teaching is no longer necessarily related to a particular level of language. Some major robot vendors are committed to increasing the capability of teach pendant programming, including some features of high-level languages. Even as it exists today, teach pendant programming is adequate for a wide variety of tasks. The vendors think that this fact, coupled with its simplicity, outweighs its disadvantages. A current issue in robot programming is whether teach pendant or off-line programming is the better method of programming robots. This issue is explored in more detail in Section 3.5.1.

1.6.2 Off-line Textual Programming

The disadvantages of teach pendant programming led to the development of off-line textual programming languages such as VAL, KAREL and AML. Sections 1.3 and 1.4 state that there is a difference of opinion as to the best approach for developing these languages. RAPT, MCL, LAMA, and others are based on the APT NC language. VAL, KAREL and others are based on extensions to high-level computer languages such as BASIC and PASCAL. Still yet another approach is to develop a new language from the ground up as discussed in Section 3.5.2. This section covers the general issues involved in off-line programming techniques.

There are several advantages to off-line robot programming as opposed to on-line programming with a simple pendant. The major ones are listed below:

1. The robot is not rendered unproductive during programming time. In small and medium batch operations, this is extremely important.

2. Off-line programming allows for easier incorporation of vision and force sensors, which is required in assembly tasks.

3. Off-line languages allow synchronization of external equipment to be handled quite efficiently. In many cases a robot program becomes the master program in a manufacturing cell, after extensive off-line simulation and optimization.

4. Branching and looping constructs of a language allow programs to perform customized error handling. For example, a limit switch in the gripper can be checked after picking up a part. If the part is not present, program control can be branched to a specific routine written to handle that kind of error condition. Depending on the level of language generated by the teach pendant, an on-line programming system may also have this capability.

5. Repetitive tasks such as palletizing can be programmed with relative ease by using macro or subroutine capabilities. Newer teach methodologies also have this capability.

6. A library of subroutines can be developed for use in future programming tasks. Previously programmed solutions can be utilized to write new programs instead of coding every program from scratch.

7. Several programmers can simultaneously work on various parts of a large program, and then their individual modules can be linked together.

8. Off-line programming clearly separates the programming environment and the operating environment. Sophisticated programming tools can be used to aid in program development, while not burdening the computational capabilities of a controller.

Although there are many advantages to off-line programming there are some major disadvantages. These are outlined below:

1. It is extremely difficult to visualize a robot path in three-dimensional space. Trying to determine the reachabilities, proper orientations, and collision-free paths poses significant problems. This is much easier to do with teach pendant programming, where the robot path is created as the program is written.

2. Limitations of robot accuracy may result in execution errors. The accuracy of a robot is the ability to position the end-effector at a specified target point, relative to some absolute external frame of reference. The deviation between the point moved to, and the point actually desired, is a measure of the accuracy. Inaccuracies result from loading conditions and from the configuration of the manipulator in the workspace. Tight absolute positioning tolerances may not be achievable with a program generated strictly off-line unless sensors are used to position the robot with "terminal guidance."

3. The programmer cannot anticipate all the exact locations and orientations of equipment in the workspace. Therefore, all programmed endpoints are likely to be in error by some amount. The program generated off-line may need to be "fine tuned" on the shop floor prior to production, perhaps using a teach pendant, or performance may suffer. The alternative of massive sensor integration may not be economically justifiable.

4. Program modules that work in isolation may not work when linked together and run on the robot. This is often a result of the dynamics of the robot.

5. The exact path that will be taken by the robot is not often known at programming time. This is because the robot controller generates its own intermediate points as a function of initial orientation, final orientation, load, speed, and sensor input. Also, the robot may stop in the middle of executing a move, reorient itself, and then continue. This is a function of internal software which attempts to avoid having the robot exceed travel limits on the various linkages, or stretching the manipulator umbilical cord.

6. The full syntax of a language may be very difficult to learn if one is not a computer programmer.

The off-line programming technique can produce programs at any of the three levels. It is not practical to write servo level programs in an off-line mode, so this is seldom, if ever, done. Most off-line textual programming languages are at the manipulator level, although some do have a little task level capability. Some manipulator level languages are unstructured, some are structured, and some are functionally-oriented. Typically, robot controllers use the manipulator level languages as input. Software internal to the robot controller, i.e., a translator or interpreter, then breaks down these programs in several steps until the actual servo signals to the drives are generated.

The problems with off-line manipulator level programming are the driving force behind the development of task level languages. At this level, programs can be written in an off-line mode by having the programmer specify tasks to be performed. The problems associated with collision avoidance, orientation, and so forth are solved by a computer using internal models of the world. There are significant problems to be overcome before this becomes a reality (Section 5.3). It is likely that first task level languages will be processed in an off-line system where they will be decomposed (translated) into manipu-

in an off-line system where they will be decomposed (translated) into manipulator level languages for use by the robot controllers. Eventually, task level languages themselves may be input directly into the controllers, perhaps even using an on-line teach pendant.

1.6.3 Augmented Teach

Beginning with the development of the first commercial robot programming language (VAL) in the 70s, vendors have often combined the method of textual programming with a teach pendant session to produce an executable robot program. This combination of techniques can result in a flexible and economical approach which we will refer to as "augmented teach" as opposed to the simpler teach of Section 1.6.1.

Augmented teach programming results in the generation of two sets of data. The first is a simple text program file composed of a sequence of statements similar to those generated by off-line textual programming. Positions in the robot workspace are referred to symbolically, rather than with numerical coordinate values. The second data set is a file containing a listing of the position symbols used by the text program, associated with coordinate values to be used at execution time. This data set is obtained by manually jogging the manipulator to the actual points in space corresponding to the named positions, and then storing the actual joint angles or coordinates of the manipulator.

Individually, the data sets are insufficient to run the robot. Placed together, by a compiler or loader, an executable robot program results. There are two advantages to this augmented teach method. First, the text motion program can be developed off-line and simulated to some degree. Second, the program can be reused with different coordinate data sets, corresponding to different part geometries.

The implementation of VAL and CIMPLER and many others requires the coordinate data set to be created during a teach session with a robot. Other systems, such as the ASEA Off-line Programming System, permit the coordinate file to be generated without robot interaction if desired, possibly using coordinate values from a CAD system, and thus can be programmed entirely off-line. Their system allows a third parameter data set to be incorporated off-line as well (7).

The key value to the augmented teach method lies in the separation of program structure and geometric data values. Consequently, a given program may be used without modification to manipulate different parts, simply by changing the coordinate value data sets.

1.6.4 CAD/Graphics Systems Programming

Since one of the major problems with off-line textual programming is in visualizing robot motion, the natural places to turn to in solving the problem are graphics systems and computer aided design (CAD) systems. There are several types of systems here. The first is a graphics system used to read in a manipulator level program and, based on a model of the robot and surrounding equipment, presents a simulation of the robot motion. The programmer can then check for collisions, proper orientations, reach, and so forth. The other type of system runs an application program in which the programmer interacts with a graphic display of the robot in order to generate the program. Motion, for example, may be programmed by indicating endpoints with a light pen. Note that in the latter system the programmer actually generates the program with the aid of the simulation, while in the former system the program is completed prior to the simulation. Note also that the latter system is robot independent. The same applications program can be used to generate programs for several different types of robots. Postprocessors are used to convert the application program output into robot specific languages. The application program itself may facilitate programming at the task or manipulator levels, but the output from the postprocessor is typically at the manipulator level. In either system, models of the robot, equipment, or parts may be extracted from a CAD data base. Another method of programming is to write a manipulator level program using a text editor and allow object descriptions to be supplied by the CAD data base. The variety of uses for CAD and graphics systems is illustrated by the commercial products developed by McDonnell Douglas Automation (Section 2.5.2) and others. Since there are no clear delineations between these systems, we shall collectively refer to their use as CAD/Graphic systems programming.

The advantages to having a simulation system are that it provides a visual aid for analyzing motion (8), time studies can be performed (9), and workcells can be designed (10) by trial and error methods. This can result in enormous time savings in robot program development and cell layout. The economics of small batch assembly are heavily impacted by the cost of the robot and auxiliary hardware and the assembly cycle time (9). While hardware costs may continue to drop, immediate reduction of assembly costs is afforded through time studies designed to optimize not only the work cell layout but also the robot program design.

A further advantage to the interactive systems is that the programmer need learn only one application program. There are, however, some significant limitations to these systems (3). First, time studies are only a rough approximation. This is because the systems assume instantaneous accelerations and deccelerations. There is no modeling of slop, backlash, or overshoot in the joints. Simulation of these effects would require a complex dynamic model of the robot and these are not available from the vendors. Second, most systems use wireframe models, and therefore do not have automatic interference checking. The programmer must run through the simulation from different views and visually check for collisions. Wireframe displays of complex scenes are sometimes difficult to interpret. Solid modelers are available at a reasonable cost, but are very slow. Third, sensory inputs to the robot, such as vision and force, are difficult to simulate. Finally, many of the problems present in off-line textual programming are still present. The simulation is not an accurate geometric representation of the real world, so motion commands must be "fine tuned" on the floor. Also, the software that causes a reorientation of the robot in the middle of a move is not simulated. As with off-line textual programming, the final debug of the program must occur on the shop floor.

Task level programming will rely heavily on CAD models for information needed to decompose tasks into the appropriate motions. CAD-based programming techniques will therefore probably be the most popular method used to generate task level programs. Task level programs could be generated solely by textual

means, but this would require describing all of the features of the robot, the parts, and the environment explicitly with textual commands. When it is possible to store libraries of models in CAD, it is unlikely that the textual approach will be taken.

1.7 ROBOT PROGRAMMING PROCESS

The program development cycle for robotics is markedly different from computer programming, for several reasons. It is worthwhile to compare and contrast the process of programming in these two areas. From this comparison, insight may be gained into the problems of robot program development. Figure 3 illustrates the off-line robot programming process.

In the most general sense, robot programs function to manipulate elements in the immediate environment, according to dynamic real-time constraints imposed by the process or material involved. Computer programming, by most definitions, is concerned with the manipulation of symbolic data according to constraints based on the nature of the data or manipulation methods. In short, robot programs must be concerned with the real-time control in a relatively unconstrained environment, while computer programs for the business world ordinarily operate with few timing constraints over well defined sets of data. As an obvious consequence, error handling comprises the largest part of robot programming, while input and output statements play a relatively small role. The converse is true of business-oriented programming, which primarily functions to process input data files and create output files.

A second reason for the difference between computer and robotic programming lies in the physical division between the programming environment and operating environment. The programming environment is where the programs are created, revised, and sometimes simulated. Simulation is the only mechanism whereby the functionality of the program may be checked before transporting the program to the operating environment. In the operating environment, the actual semantic debuggging session occurs.

Note that in computer programming, after a program is checked for proper syntax and converted into an executable form, it is tested on the same system with sets of good, and then bad, data designed to test specific portions of the program. The resulting output file may be rigorously checked for errors which can be traced back to program errors. This develop/test/rewrite cycle

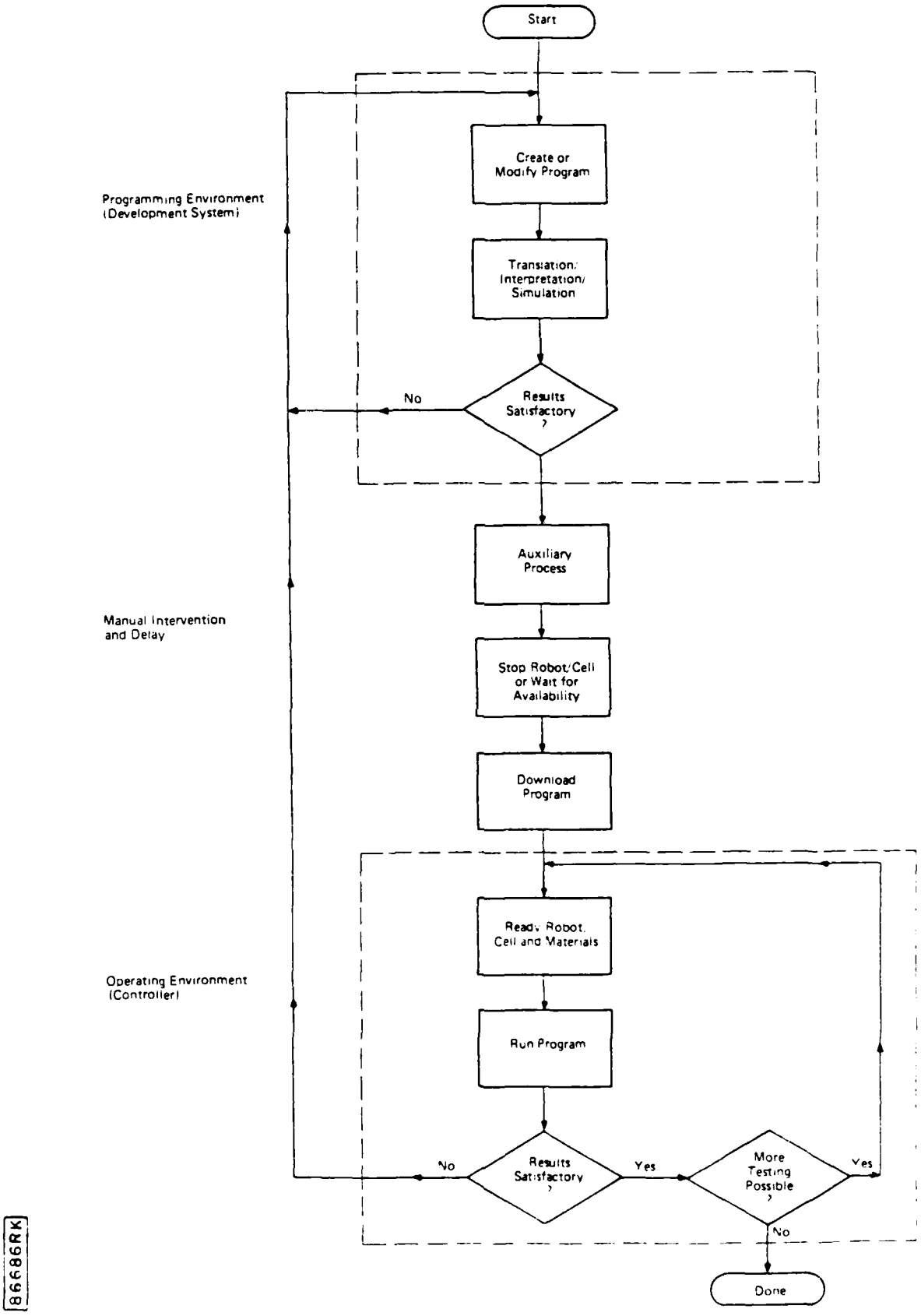


Figure 3. The off-line robot program development cycle. (Source: IITRI).

may occur many times before achieving acceptable results. In off-line robot programming, after a similar conversion step, the program is transported to a different system having a markedly different architecture and set of resources. It is then tested by allowing the robot to physically perform the programmed steps. The manual, and often unrecorded, observation of the test then serves to motivate changes to the original program, accomplished back on the development system. The number of testing sessions available for a robot programmer is limited by the costs involved in using the robot and consumable materials. Beyond a few such iterations, development costs become prohibitive.

To summarize, the robot program development cycle is different from that of normal computer programming, because of differences in the program functionality, development and operating environments and testing methodology and costs. The capability of a robot programming system can be evaluated using criteria developed for computer programming, as is shown in later sections. The program development cyle, leading to a measure of programmer productivity, can be discussed in terms of the process outlined so far. The time lag between program creation or revision and on-line testing can be reduced by using a higher level of language and more powerful tools in the programming environment. The number of develop/test iterations necessary to achieve an acceptable program is partly a function of the program design. A stronger design is achievable through language features, combined by an experienced programmer, to utilize sensors for the detection and analysis of error conditions. The number of iterations and the compactness of the development/test cycle form a measure of the cost of robot program development. The issues involved in the programmability of a system are thoroughly addressed in Section 3.

1.8 SUMMARY

Two robot programming issues have been identified thus far. There are two general philosophies to robot programming approaches: teach pendant programming and off-line programming. Among those who consider off-line programming to be the proper approach, there is further disagreement as to whether the language should be an extension to existing general purpose computer languages, an extension to NC languages, or a totally new language. Teach pendant programming was the first technique and is adequate for a variety of tasks. Some vendors are striving to increase its capability either by

increasing the power of the pendant or using the pendant to fine tune an existing off-line program. Off-line textual programming languages offer more power, but are considerably more complex to learn and have other drawbacks. CAD/Graphic techniques have been developed to address some of these drawbacks.

Whatever technique is used to program a robot, the output from the technique is a program that resides at one of three levels: task level, manipulator level, or servo level (see Figure 1). Task level programming is still in the future, but some languages have begun to incorporate task level capabilities. Programming at higher levels is easier for the programmer, but still requires that the program be decomposed into the servo level commands necessary to move the robot. This can be done in an off-line system through translation or in the robot controller itself. To incorporate this capability into the robot controller requires fast and powerful computational ability at a reasonable cost. Current trends in hardware seem to indicate that this is a realistic expectation.

Section 2 of this report reviews the current programming languages. Section 3 investigates the requirements of a robot programming language, look at the basis of comparisons that can be used when evaluating languages, and compare a selection of languages.

2. CURRENTLY AVAILABLE ROBOT PROGRAMMING LANGUAGES

2.1 LANGUAGES IDENTIFIED AND CLASSIFIED

Table 1 is a list of many of the higher order robot languages that were identified during this effort. This list was prepared with an open mind and, if in doubt, a language was listed. Note that some of those listed, i.e., MHI (1960) and WAVE (1970), formed the basis of other languages and had a world-wide impact on robot language development, but are not to be considered as currently available languages. Only those languages that have been accepted as being applied to robotics were listed. For example, Ada has been suggested as a base for a robot language (11) (12) (13), but was not included because it has not yet been accepted as such.

Initially, we identified 89 languages. This list was reduced to include only those cited in more than one publication. It was assumed that those that had only one reference were of transitory interest and were either dropped as developments or incorporated into other efforts.

The literature can be confusing as to the popularity of any one language because the frequency of publication is based more on the marketing approach of the developer than on the merits of the language. University developed languages may be the subject of several papers, but may never have been used commercially. On the other hand, some commercial vendors are not prone to publish papers on their current R&D efforts. It is difficult, particularly among the more recent developments, to determine which languages are likely to survive in the commercial marketplace. An analysis of several languages that seem to currently have major support is in Section 3.4.

The identification of the developer is not always as clear as one would like. In fact, in some cases the literature was contradictory in this respect. Individuals have written papers on a language after they left the company or institution in which it was developed. In other cases development was done under subcontract or jointly between several organizations. Also, many of the languages are largely based on other languages, so the reported developer may have only modified or extended to a small degree a language developed elsewhere.

TABLE 1. HIGHER ORDER ROBOT LANGUAGES (page 1 of 7)

A Partial Listing

Name	Meaning	Developer	Year	Basis	Robots on Which Used	Status	Type of Language
AL		Stanford, USA	1974	Concurrent PASCAL	-1	Development	Task and manipulator level
AML	A Manufacturing Language	IBM, USA	1977	ALGOL, APL, LISP	IBM Robots	Commercial	Manipulator level
AML/E	A Manufacturing Language/ Entry	IBM, USA	1982	ALGOL, APL, LISP	IBM Robots	Commercial	Manipulator level
AML/V	A Manufacturing Language/ Vision	IBM, USA	1982	ALGOL, APL, LISP	IBM Robot System	Commercial	Machine vision
AML/X	A Manufacturing Language/ Experimental	IBM, USA	1986	SmallTalk, APL, LISP, AML	-1	Development	General purpose manufacturing/CAD
AML/2 purpose	A Manufacturing Language/2	IBM, USA	1986	AML/X	IBM 7575/6	Commercial	General purpose and Manipulator
AR-BASIC		American Robotics	1982	BASIC	AR Robots	Development	Manipulator Level
ARL	Assembly Robot Language	Hitachi, Japan	1983	PASCAL	Hitachi	Development	Task Level servo-control level
ARLA	ASEA Robot Language	ASEA, Sweden	1982	-2	ASEA	Commercial	Manipulator
AUTO-PASS	Automatic Parts Assembly System	IBM, USA	1977	Model based	-1	Development	Task level

-1 Language under development for unspecified robot.
-2 Language with no identifiable basis.

TABLE 1. HIGHER ORDER ROBOT LANGUAGES (page 2 of 7)
A Partial Listing

Name	Meaning	Developer	Year	Basis	Robots on Which Used	Status	Type of Language
CIMPLER		GCA, USA	1985	-2	GCA	Commercial	Manipulator level
DARL		Seiko		-2	Seiko	Commercial	
EMILY	ML Extended	IBM, USA	1975	FORTRAN	-1	Development	Manipulator level
HARL2	Hirata Assembly Robot Language	Hirata, Japan	1983	-2	Hirata	Commercial	Manipulator level
HELP		DEA, Italy	1979	PASCAL Based	PRAGMA A3000 family of	Commercial	Manipulator and servocontrol
HIGH		University of Tokyo, Japan	1983	FORTRAN	-1	Development	Manipulator level
JARS		JPL, USA	1979	PASCAL Based	Puma 600	Development	Manipulator level
KAREL		GMF Robotics, USA	1985	PASCAL Based	GMF Robots	Commercial	Manipulator level
LERNA		Inst de Microtechniques de l'ERFL Switzerland	1984	-2	-1	Development	Task level programming for multiple robot systems which uses teach mode for the individual robots

TABLE 1. HIGHER ORDER ROBOT LANGUAGES (page 3 of 7)
A Partial Listing

Name	Meaning	Developer	Year	Basis	Robots on Which Used	Status	Type of Language
LM	Language Manipulation	University of Grenoble, France	1980	PASCAL Based AL Like	Renault TH8 and Kemlin Robots	Development	General manipulator level
LMAC	Modular Command Language for Industrial Robots	Several Organizations France	1984	PASCAL Based	-1	Development	Task level program for multiple robots
LPR	Language de Programmation pour Robots	Renault and University of Montpellier France	1983	-2	Renault, Cybotech and ACMA Robots	Commercial	General purpose manipulator and servocontrol
MAL	Multipurpose Assembly Language	Milan Polytechnic Italy	1979	BASIC Based	Research Robot	Development	Manipulator and servocontrol
MAL	Multipurpose Assembly Language	Milan Polytechnic Institute, Italy	1983	FORTRAN	Sigma	Development	Task Level Manipulator Level
MAPLE		IBM, USA	1976	PL/I	-1	Development	Manipulator level
MCL	Manufacturing Control Language	McDonnell Douglas USA	1979	APT Based	Cincinnati Milacron T3	Development	Task level servocontrol level Graphic simulation
MHI	Mechanical Hand Interpreter	MIT, USA	1960	-2	-1	Development	Manipulator and servocontrol level

TABLE 1. HIGHER ORDER ROBOT LANGUAGES (page 4 of 7)
A Partial Listing

Name	Meaning	Developer	Year	Basis	Robots on Which Used	Status	Type of Language
MINI		MIT, USA	1972	LISP Based	-1	Development	Manipulator and servocontrol levels
ML	Manipulator Language	IBM, USA	1973	BASIC	IBM Robots	Development	Manipulator and servocontrol levels
MYBASIC		Hobart				Commercial	
PAL	Portable AL	Purdue, USA	1978	AL Based	-1	Development	Servocontrol
PAM		Remak	1982		Remak	Commercial	Manipulator level
PASLA	Programmable Assembly Robot Language	NEC, Japan	1983	-2	-1	Development	Manipulator Level
PASRO	PASCAL for Robots	Biomatic, West Germany	1983	PASCAL Like AL Approach	Microrobot	Development	Manipulator
RAIL		Automatix, USA	1982	PASCAL Based CAD/CAM	Hitachi, AID 800	Commercial	Manipulator level Vision system compatible for any robot

TABLE 1. HIGHER ORDER ROBOT LANGUAGES (page 5 of 7)
A Partial Listing

Name	Meaning	Developer	Year	Basis	Robots on Which Used	Status	Type of Language
RAPT	Robot APT	University of Edinburgh, Scotland, UK	1978	APT Based	-1	Development	Task based feeding into manipulator language
RCCL	Robot Control "C" Library	Purdue, USA	1983	C Based	Puma	Development	Task Level
RCL	Robot Command Language	RPI, USA	1983	Assembler		Development	Manipulator Level
RISE		Prab, USA		-2	Prab G-series	Commercial	Manipulator Level
ROBEX	ROBoter EXapt	University of Aachen, West Germany	1980	APT Based FORTRAN Implementation	Commercial used by Sieman for various robots	Commercial	General purpose manipulator language with limited servocontrol level and graphic input
ROBOCAM		Prab, USA	1985	-2	Prab G-Series Robots	Commercial	Manipulator level with CAD interface
ROCOL		Leningrad Polytechnic Institute, USSR	1983	-2	LPI-2	Development	Task Level Manipulator Level

TABLE 1. HIGHER ORDER ROBOT LANGUAGES (page 6 of 7)
A Partial Listing

Name	Meaning	Developer	Year	Basis	Robots on Which Used	Status	Type of Language
ROMPS	Robot	CAM-I, USA JAIS, Japan	1984	-2	-1	Development	Task Level
ROPS	Robot Off-Line Programming Systems	Cincinnati Milacron, USA	1985	-2	Cincinnati Milacron T700s and T800s	Commercial	Manipulator level
RPL	Robot Programming Language	SRI, USA	1980	LISP, FORTRAN	Puma 500 Unimate 2000A	Development	Task Level
RSS	Robot Servo System	University of Illinois, USA	1983	-2	-1	Development	Servocontrol level
SERF	Sankyo Easy Robotic Formula	Sankyo Seiki, Japan	1983			Commercial	Manipulator Level
SIGLA	SIGma LAnguage	Olivetti, Italy	1978	AL Based	Sigma Robots	Commercial	Manipulator level
SRL	Structural Robot Language	University of Karlsruhe West Germany	1983	PASCAL Based Portable AL extension	-1	Development	Task level, manipulator level and servocontrol level

TABLE 1. HIGHER ORDER ROBOT LANGUAGES (page 7 of 7)

A Partial Listing

Name	Meaning	Developer	Year	Basis	Robots on Which Used	Status	Type of Language
T3		Cincinnati Milacron, USA	1978	-2	Cincinnati Milacron T3	Commercial	Manipulator level
T-CAM	Thermwood			-2	Z-8000 Controller	Commercial	
TEACH		CIT, USA	1979	-2	Bendix PACS	Development	Manipulator level
VAL	Versatile Assembly Language	Unimation, USA	1975	Basic Like Wave Based	Puma 2000 Unimates 4000 Unimates	Commercial	Manipulator level with binary servo-controls
VALII	Versatile Assembly Language II	Unimation, USA	1984	VAL	Unimate/ Westinghouse	Commercial	Manipulator and servocontrol level
VML	Virtual Machine Language	Milan Polytechnic CNR Ladseb of Padova, Italy	1980	-2	-1	Development	Manipulator level with some task level capabilities
WAVE		Stanford/Purdue, USA	1970	-2	-1	Development	Manipulator Level

Source: IIT Research Institute

The year listed is the earliest year associated with a specific language. Again, the literature is contradictory. In some instances the language may have been fully developed and in use prior to the year listed. In other instances development of the language may have been under way or just getting started in the year listed.

The basis for a given language is listed from comments, often by the developer of the language itself. The distinctions here are modest at best and for many of the languages no basis is listed. This does not mean that it was a totally new approach, but only that no reference to the base language was found.

Where possible, the actual robot arm on which the language was run is listed. This does not necessarily mean that the language runs efficiently or is commercially available on the arm, only that it has been tested on that arm. Robot vendors are the best source of information for determining what languages are available on their robots. Table 2 lists some commercially available languages and the robots on which they run.

When a language is in commercial use, it is listed as such. It was assumed that simply offering to sell a language does not make it commercial, unless it has actually been installed by user companies.

The language level is indicated in the last column. Although no task level language actually exists, those languages having some task level capability are listed as such to distinguish them from purely manipulator level languages. CAD/Graphics systems are also listed.

Several references (1), (5), (14), (15), (16), (17), (18), (19) briefly discuss some of the languages listed in Table 1, and then go on to compare them on several features. Other references are devoted entirely to a single language. This information has been compiled and is presented in the sections below. The information is not intended to be detailed, but gives a flavor of the issues explored in robot language developments and how the languages have evolved. The summaries of languages in the literature tend to hit on a few key points for each language, which may at first seem to be a bit

TABLE 2. CURRENTLY AVAILABLE COMMERCIAL LANGUAGES

ROBOT VENDOR	LANGUAGE	CONTROLLER ON WHICH LANGUAGE RUNS	VENDOR-SPECIFIC LANGUAGE
AMERICAN ROBOT	AR-BASIC	MERLIN	YES
ASEA	ARLA	S-2	YES
CINCINNATI MILACRON	ROPS	all models	YES
CYBOTECH	-1	RC6,RC7	-
DE VILBISS	-2	-	-
GCA CORP.	CIMPLER	CIMROC	YES
GENERAL ELECTRIC	-3	-	-
GMF ROBOTICS	KAREL	RF	YES
HIRATA CORP.	HARL-2	HAC Cell Controller	YES
HOBART	MYBASIC	-	YES
IBM	AML	7532	YES
IBM	AML/2	7575,7576	YES
MOBOT	-4	ALLEN BRADLEY PLC GE PLC	NO
PRAB	ROBOCAM/RISE	G-SERIES	YES
REIS	ROBOT-Star	G-70 LR-30 L4-70 V-15 H-15 H-30	YES
SEIKO	DARL, DARL-II DARL	DETRAN II-180	YES
THERMWOOD CORP.	T-CAM	Z-8000	NO
UNIMATION	VAL, VAL-2	all models	NO

[1]-Menu-driven user interface which allows off-line programming through teach pendant.

[2]-Teach pendant programming only.

[3]-No longer support Help language.

[4]-Ladder logic programming.

Source: IIT Research Institute

obscure. For example, one language description talks about coordinate system transformations, another talks about a different approach to motion specification, another mentions multitasking, and so forth. These issues are explored in detail and put into the context of robot programming in Section 3.2. At this point, it is sufficient to note what issues are being addressed and what approaches have been taken.

2.2 SERVO LEVEL LANGUAGES

These languages are the result of older teach pendant programming methods. They are not languages in the sense of computer languages, but are a means of programming at the servo level. We would not consider these to be higher order languages and therefore do not discuss them in this section. Included here are T3 (Cincinnati-Milacron), Funky (IBM), and AR-SMART (American Robot). As noted in Section 1.6.1, however, some vendors have committed themselves to improving the capabilities of teach pendant programming. Current R&D efforts in this regard are within the scope of this paper and are noted in Section 5.1.

2.3 MANIPULATOR LEVEL LANGUAGES

Manipulator level languages are typically written using editors of some sort, but may also be output by newer teach pendants or a CAD/Graphics programming system. In the latter case, the language itself is discussed here and the programming system is discussed in Section 2.5. The languages have been separated into those developed by universities and research institutes and those developed by commercial vendors. Within each of these groups, they are listed in approximately chronological order. Usually, languages developed within the research environment are not meant for commercial release; they are developed with a particular research goal in mind. They do, however, influence the development of commercial languages.

2.3.1 Universities and Research Institutes

WAVE. Developed in 1973 as an experimental language for research purposes at Stanford AI Laboratory, the objective of WAVE was to find the limitations of robotic theory, rather than to perform manufacturing tasks. (14)

the use of subroutine calls from a general purpose program. Applications cited for this language are material handling, inspection, and assembly. (14) (5)

VML. This was developed at Milan Polytechnic, Italy in 1980. It was intended as an intermediate language between artificial intelligence systems and the robot. It is used to transform points in Cartesian space into joint space (manipulator to servo level). (1)

LENNY. Developed in 1982 at the University of Genoa, Italy, the key issue of LENNY is its functionality. It was intended as a language to be understandable by humans and to be powerful enough to express complex chains of actions, processes, and concurrent computations. (1)

LPR. Developed in 1983 as a joint venture by Renault and the University of Montpelier in France, this language is based on definitions of state graphs and transitions between states. The state graphs are defined with a hierarchical basis such that all graphs at the same level are executed in parallel. The language supports 24 I/O ports for sensors and synchronization with external equipment. (1)

MAL. Milan Polytechnic in Italy developed two versions of MAL, one is based on BASIC (1979) and the other on FORTRAN (1983). The latter version ran on a Sigma robot while the former ran on a research robot. The language can be used to program a two-arm Cartesian robot. It features parallel execution of tasks, subroutine calls with argument lists, and sensor interfaces. (1)

RCCL. Developed at Purdue University in 1983, the approach here is to embed robot commands in a library of routines written in the C language, making use of the UNIX operating system. The objectives of the language were to provide rich manipulator control commands, flexible data processing, and standardization potential. (17)

RCL. This is a command-oriented motion control language developed at Rensselaer Polytechnic Institute in 1983. (5)

SRL. This was developed in 1983 as part of a standardization project in West Germany, at the University of Karlsruhe. The language is based on Portable AL and PASCAL, and the user program is translated into a robot independent code called IRDATA (see Section 3.5.5). SRL data types include those of

PASCAL and artificial intelligence abstract data types. Instructions can be executed sequentially, in parallel, in a cyclic way, or in a delayed way. Both straight line and circular motion are possible. (1)

LMAC. This was a coordinated effort among several French organizations, and was developed at the University of Besancon in France. It was designed to ensure safe control of mechanical devices in an automated cell. The key issue is its modularity, which is based on the implementation of abstract data types. Program modules written in a variety of languages may be referenced by an LMAC program, permitting a simple interface to existing programs. (1)

2.3.2 Commercial Vendors

VAL. This was the first commercially available language, released in 1978 by Unimation. It was developed by a Stanford graduate familiar with AL, under the constraint that it could be run on a minicomputer. Resembling BASIC, the language was user-friendly to nonprogrammers. It was designed for general purpose manipulation only, and had very limited data processing capabilities. (17)

ML. This language was developed by IBM in 1973, but was never released commercially. (25)

EMILY. Based on ML, this language was developed by IBM in 1975, as an extension to ML. (25)

SIGLA. Developed by Olivetti in Italy in 1978, this is a software system with a supervisor for interpreting a job control language, a teach module for teach by guiding, an execution module, and an editor. It is really a programming environment for use on Sigma robots. (1)

AML. This was released in 1982 by IBM. AML was designed to be a general language for manufacturing equipment control, but its use to date has been mostly for robotics. The philosophy behind the design of this highly structured language is to provide low level and powerful primitives which the user then builds into libraries of routines. This results in a language that is extremely flexible and powerful, but requires some programming skill. The language provides for multitasking, host system communications, user-defined reference frames, program control features for structured programming, and advanced sensory interface capability (14) (16) (17) (19) (22). AML is discussed in more detail in Section 3.4.4.

AML/E. This is used on the 7535 and 7540 Scara-type. It is an entry level subset of the AML language. (16)

AML/V. This is a package of AML subroutine providing vision capability. (22)

HELP. This language was introduced in 1982 by General Electric under license from Digital Electronic Automation in Italy. It is a structured language with a PASCAL-like syntax. Robot motion is described in terms of rectangular coordinates. It has a powerful I/O capability, and allows for defining and activating multiple tasks which communicate with global flags. (1) (14)

MCL. This language was developed by McDonnell Douglas under an Air Force ICAM project and was introduced in 1981. It was written for use in manufacturing cells, is APT-based, and uses a CAD data base as a source for geometric information. The language supports vision, image modeling, process coordination, real time conditional logic, multiple coordinate systems, macros, and compile-time language extensions. (14) (19)

RAIL. Developed by Automatix in 1981 to control their AID 800 robots for inspection, arc welding, and assembly, this language is loosely based on PASCAL. It is unique in that it was designed to control both robots and vision systems instead of having vision added as an extension. Motion types supported are straight line, coordinated axis, and oscillating paths (for welding) (14) (16) (19). Reviewed in more detail in Section 3.4.2.

ARLA. ARLA is the ASEA Robot Language, released in 1982 for their line of IRb robots. This language is available for on-line program development with a teach pendant or off-line for VAX and IBM computers. The ASEA Off-Line Programming System allows for the creation of manipulator level programs in ARLA, which is an unstructured language having constructs for motion, sensor interaction, communication, math and logic functions, and application specific entities for welding and vision. Coordinate data sets may be generated off-line with an editor, or on-line during a teach session.

PASRO. Developed in 1983 by Biomatics in West Germany, this is a PASCAL-based language with data types and procedures added for robot-specific tasks. Its development was based on experience with AL. (1)

CIMPLER. This is a manipulator level language which runs on the CIMROC controller of GCA. It has some sophisticated features for motion control, concurrent execution and I/O, which shows the designers clearly had an industrial manipulation viewpoint. (20)

VAL-II. This language was introduced in 1984 by Adept for use on Westinghouse/Unimate robots. Whereas VAL was weak in its data processing capabilities, VAL-II offers structured constructs, arithmetic functions, external path modifications, and communication support. A process control task can run concurrently in the background during motion execution. (15)

AR-BASIC. This language was released in 1984 by American Robot for use on their robots. It is based on BASIC, and supports discrete I/O, mathematical capabilities, and programming flow control. (16)

KAREL. Introduced in 1985 by GMF Robotics, this is a PASCAL-based language offering structured logic constructs, high speed communications support, vector and array structures, process control capability, vision system integration, and textual I/O. The language supports point-to-point motion control and linear and circular interpolation (16) (21). It is reviewed in Section 3.4.3.

AML/X. This is a general purpose programming language by IBM for manufacturing and computer aided design. It is a major revision of AML with sophisticated data abstraction capabilities (22). It is reviewed in Section 3.4.4.

2.4 TASK LEVEL LANGUAGES

As noted earlier, no true task level language exists. Different languages have been mentioned by different authors as having some task level capability. Among these are AUTOPASS, LAMA, AL, RAPT, and ROBEX. RAPT and AUTOPASS are the most often cited. The success with task level implementations is limited to date. AUTOPASS and LAMA were defined, but incompletely implemented. AL had task level instructions, but they weren't implemented. There are many problems to be solved before task level programming is a reality (see Section 5.3), but it is worthwhile to note the efforts to date and the lessons that have been learned.

2.4.1 University and Research Institute

RAPT. Introduced by the University of Edinburgh in 1978, this is an APT-based language in which tasks are described in terms of objects, relations between objects, and motion of objects. A RAPT program consists of a description of the parts involved, the robot, the workstation, and an assembly plan. The plan is a list of geometric relations expressing what should be true at each step. The language is independent of the robot type, and a post-processor has been developed to generate VAL programs from RAPT source code. (1) (24) (23)

2.4.2 Commercial Vendor

AUTOPASS. Partially implemented by IBM, this language focuses at the object level where motion is described at a high level. (25) It is similar to RAPT, in that programs have the look of Assembly Instruction Sheets. (23)

2.5 CAD/GRAPHIC SYSTEMS

2.5.1 University and Research Institute

GRASP. This is a CAD-based programming system developed by the Rensselaer Polytechnic Institute. It ran on a PRIME 750, and post-processors were written to translate trajectory information into robot specific languages for Cincinnati Milacron T3 robots and the PUMA 600 (16).

IGRIP. This was a simulation package developed at Cornell University for studies in the kinematics and dynamics of robot motion, and for time studies. It ran on a VAX 11/780. (16) (26)

GRIPPS. Developed at the Michigan Technological University as a programming tool, this was expanded to perform motion simulation and dynamic behavior analysis. Initially run on mainframe, efforts are being made to develop a microcomputer based system. GRIPPS input files are in IGES format, and it uses the GKS library for display. (16)

2.5.2 Commercial Vendors

PLACE, ANIMATE, COMMAND, ADJUST, BUILD. These are packages developed by McDonnell Douglas Automation that perform a variety of tasks using the Unigraphics CAD system. PLACE is a simulation tool for designing and evaluating workcells. It checks positions, limits of reach, motion sequences, and cycle

times. ANIMATE reads MCL robot programs and displays the resulting robot motion. COMMAND combines commands in native robot language with motion created graphically into a robot program. ADJUST is used to calibrate off-line programs for actual workcell variations. BUILD allows engineers to generate control equations for various robot geometries. (16)

ROBOT-SIM. This software was introduced by GE/Calma, and emphasizes kinematics and dynamics. Initially used with GE products, it supports workcell development and robot motion simulation, analyzes dynamic response, and calculates dynamic path errors. (16)

2.6 CURRENT LANGUAGE USE AND TRENDS

The most commonly used languages to date are listed in Table 3. These languages have established usage in industry. Thus far, the real driver to language acceptance has been the robot vendor community. Typically each vendor has developed a language for use on its own robots, and languages have gained acceptance as a consequence of robot selection and use in industry. Note that only one of the languages in Table 3 has been developed as a robot-independent language; the others are specific to the vendor that developed them. RAIL was developed as a language specific to Automatix vision systems. Note also that one of the languages listed (T3) is servo level. The reason is that, for several years, simple teach pendant programming was the only method available on Cincinnati-Milacron robots.

Several more recent languages are not listed in Table 3. Some languages introduced at the Robots 9 show in the Spring of 1985, for example, are not listed because their applications thus far are limited and their eventual acceptance is unknown. Included here would be ROBOCAM, HERON, CIMPLER, and ROPS. We expect that some of these languages will gain wide acceptance over the next few years. With regard to the robot-specific languages, their acceptance will be dictated by the success of the robot vendors that supply them.

Languages will become an increasingly important factor in robot selection in the future. Most robots are used for relatively simple applications in which teach pendant programming is sufficient. In these applications, programming is usually a secondary consideration. As robots are used for more complex applications, however, language considerations have a significant cost

TABLE 3. SIGNIFICANT ACCEPTANCE AND USE TO DATE

NAME	SOURCE COUNTRY	ROBOT TYPES
AML and AML/E	USA	IBM
AR-BASIC	USA	American Robot
ARLA	Sweden	ASEA
HELP	Italy	Pragma GE
KAREL	USA	GMF
RAIL	USA	Automatix
ROBEX	West Germany	Various
SIGLA	Italy	Olivetti
T3	USA	Cincinnati-Milacron
VAL and VALII	USA	Westinghouse/Unimate

Source: IIT Research Institute

impact, and are likely to become a primary consideration in robot selection. In applications such as assembly, for example, program development and maintenance costs are likely to equal or exceed hardware costs. In batch job environments, the ability to develop robot programs quickly and efficiently is essential to justify the cost of the robot. In automated factories, communication (networking and file transfer) is essential. These issues all revolve around the programming language.

Another obvious trend is the increasing use of CAD/Graphics systems. In off-line programming of complex applications, simulation of robot motion is essential. As more detailed models of robots, machines, and the parts they handle are put into CAD systems, more CAD-based programming systems will be used to automatically access these models for use in program development. The proliferation of robot languages will also provide a driving force for increased use of CAD-based programming. User companies will want to establish an application program on CAD, so that all robot programs can be developed using the same language and programming environment. Postprocessors will then generate the robot-specific programs.

2.7 SUMMARY

The available programming languages were identified and classified as servo level, manipulator level, and task level. While no true task level languages exist, some incorporate task level capabilities. Many of the languages developed are a result of university and research institute efforts, and are usually extensions to existing robot or computer languages. Often, the research languages are developed to explore a particular facet of robot programming. In general, the evolution of robot languages parallels the evolution of general purpose computer languages in overall structure and capabilities.

The primary driver behind robot language acceptance and use has been the robot vendor community. Vendors have taken the results of previous efforts, both research and commercial, and have developed their own languages for use in controlling their robots. Very few of the existing languages are robot-independent.

Language considerations (capabilities, features, programming and operating environments) will become an increasingly important factor as robots are applied to more complex tasks. Program development and maintenance costs will be a significant factor in future robot applications.

CAD/Graphics systems' use for robot programming will increase in three ways. First, they will be used for a simulation tool to study work cell configurations, robot motion, and cycle time. There are many limitations to the current systems and steady improvements will be made. Second, CAD models of parts, robots, and machines will be used in robot program development. In this respect, CAD models will define coordinate systems and features to be used in executing the robot program. Finally, robot-independent applications programs will be developed to relieve programmers of the task of learning several robot programming languages.

3. COMPARISON OF ROBOT LANGUAGES

3.1 BASIS OF COMPARISON

When there is a proliferation of similar products it is natural to look for a way to compare them. The variety of robot programming languages represents a unique challenge, because the subject area is complex. Robot languages are far from a stage of maturity--various approaches are still being explored and there are many issues yet to be resolved. Also adding to the problem is the failure of the literature to make clear several important considerations and distinctions. These are:

1. The relationship between robot language and robot performance
2. The distinction between robot programming levels and robot programming techniques (see Section 1)
3. The distinction between language capabilities, language features, and the language implementation
4. The distinction between programming environment and operating environment, and their relation to the robot language
5. The distinction between language requirements at the task level and at the manipulator level
6. The distinction between evaluating robot languages and evaluating robot implementations.

3.1.1 Robot Performance

When a user is comparing robots, several considerations must be made. Overall robot performance is the most critical concern, and it is important to realize that the robot language is only a part of this. Other factors affecting robot performance are lift capacity, reach, accuracy, repeatability, and mean time between failures. No standards exist, but work is under way to try to establish some metrics by which robot performance can be evaluated.

The importance of the language relative to the application must be considered. A very simple application may require little programming and a high degree of repeatability, so the language is of secondary importance. Another application may not require very good repeatability, but involves complex

motion, significant sensor integration, and complicated error handling routines. In this case, the language is of primary concern. One application may be for a mass production environment, in which the robot may actually run only a few programs, with minor modifications, throughout its entire life. In this case, programming time is only a small portion of the life cycle, and relatively crude programming capability is often sufficient. In another application, such as a batch production environment, the robot may require frequent reprogramming. In this case, programmability is essential. The point to be made is that the importance of a robot programming language is relative to the intended application. When comparing two robot implementations, if the hardware based performance criteria are approximately equal, the language may then be the deciding factor in the robot selection.

3.1.2 Elements of the Robot Programming System

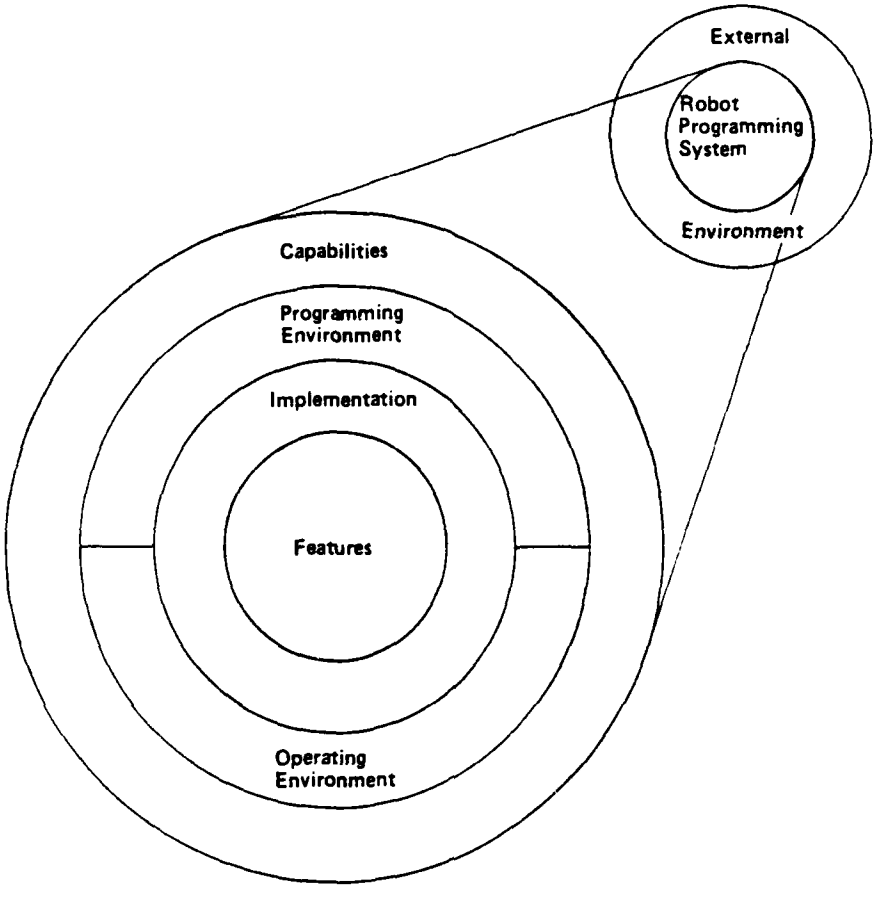
After taking into account all of the robot performance criteria, it may well be decided that the robot language is important, and that a powerful language will be needed. A natural step at this point may be to turn to the literature in search of language comparisons. There are many good language comparisons, and some are discussed in Section 3.3, but these can be very confusing if not placed within the context of an entire robot programming system.

A distinction that fails to be made in the literature is between language capabilities, language features, and the language implementation. We consider these to be three of five elements in a robot program system. The language capabilities are a measure of the overall "goodness" of a language. They are the "ilities": flexibility, extensibility, reliability, maintainability, and so on. A measure of the "ilities" is somewhat subjective, but is very important when comparing dissimilar languages. The capabilities determine the ease with which the language can be used, transported, maintained, and expanded. The set of capabilities we have chosen encompasses a very broad perspective of robot programming. Features, on the other hand, are things such as macros, conditional statements, and variable types. The features can be measured objectively--either a language has the feature or it does not. The third factor to be considered is the language implementation, i.e., unstructured versus structured and compiled versus interpreted. Thus, there are at least

three elements of the robot programming system: language capabilities, language features, and the implementation. The language implementation affects the utility of certain features in the language and characterizes the overall programming process. All of these elements are different at a conceptual level, but they are all interrelated.

A further distinction needs to be made between the programming environment and the operating environment. Essentially, the programming environment generates the robot program and the operating environment executes it. The programming environment consists of the text editor, simulation tools, CAD, subroutine libraries, and so forth. The operating environment consists of the robot and any terminals or other equipment on the shop floor. In the case of off-line programming the two environments typically exist in two separate physical locations. In the case of teach pendant programming, the two environments co-exist in the same physical location and share the same computer (the robot controller). The programming and operating environments are the other two elements of the robot programming system.

The robot programming system has been shown to consist of five basic elements: language features, the language implementation, language capabilities, the programming environment, and the operating environment. These are all integrally related. They can be considered as different conceptual layers, or views, of robot programming. The features represent a microscopic view of the language whereas the capabilities represent a macroscopic view. This is illustrated in Figure 4. Each layer is shown to be composed of a set of entities. Entities in the outermost layer are related to entities in all of the layers below. The entities in the innermost layer are the language features themselves. If an entity in the outermost layer is considered important, it will require entities in the inner layers to support it. For example, if sensor support (language capability) is required, then certain I/O constructs (features) will be needed. In Section 3.2, the requirements for a robot programming language are developed as a function of these layers. This will result in a list of language features required to support the outer layers, and can serve as a basis of comparison for robot programming languages.



86682RK

Figure 4. The robot programming system. (Source: IITRI).

Before proceeding, it should be noted that another source of confusion in the literature is the failure to distinguish between requirements of manipulator level languages and requirements of task level languages. In many of the papers, manipulator level languages are discussed and compared, and then requirements are discussed in terms of (what seems to be) task level languages. Thus far, manipulator and task level languages have been treated distinctly. In fact, the evolution from manipulator level languages to task level will be a slow, steadily progressing process involving several generations of languages. The next generation of robot languages will lie somewhere between the manipulator and task levels, incorporating elements of both. Since the literature focuses on the next few generations of languages, where the distinction between manipulator level and task level is fuzzy, their discussion of language requirements can be confusing. The following sections attempt to clear up some of this confusion.

3.2 ROBOT PROGRAMMING REQUIREMENTS

The external world, within which the robot programming system must operate, imposes requirements on it. This world can be considered as consisting of two layers: the application layer and the industrial environment layer. This is illustrated in Figure 5. In this section we correlate the demands the external industrial environment and application layers place on the robot programming system. The industrial environment is the outermost layer and consists of two components: the market and the company policy. We are not attempting here to develop the outermost layer to any level of detail, but simply to illustrate the concept. The application layer, on the other hand, has been treated extensively in the literature and will be examined in greater detail.

3.2.1 The Industrial Environment Layer

Two elements of the industrial environment layer are depicted in Figure 5. A partial list of the entities in the user's product market includes responsiveness to change, product quality and cost, delivery and production schedules and standards. All of the entities impact the robot application layer, and, in so doing, affect the inner layers of the robot programming system in the following ways.

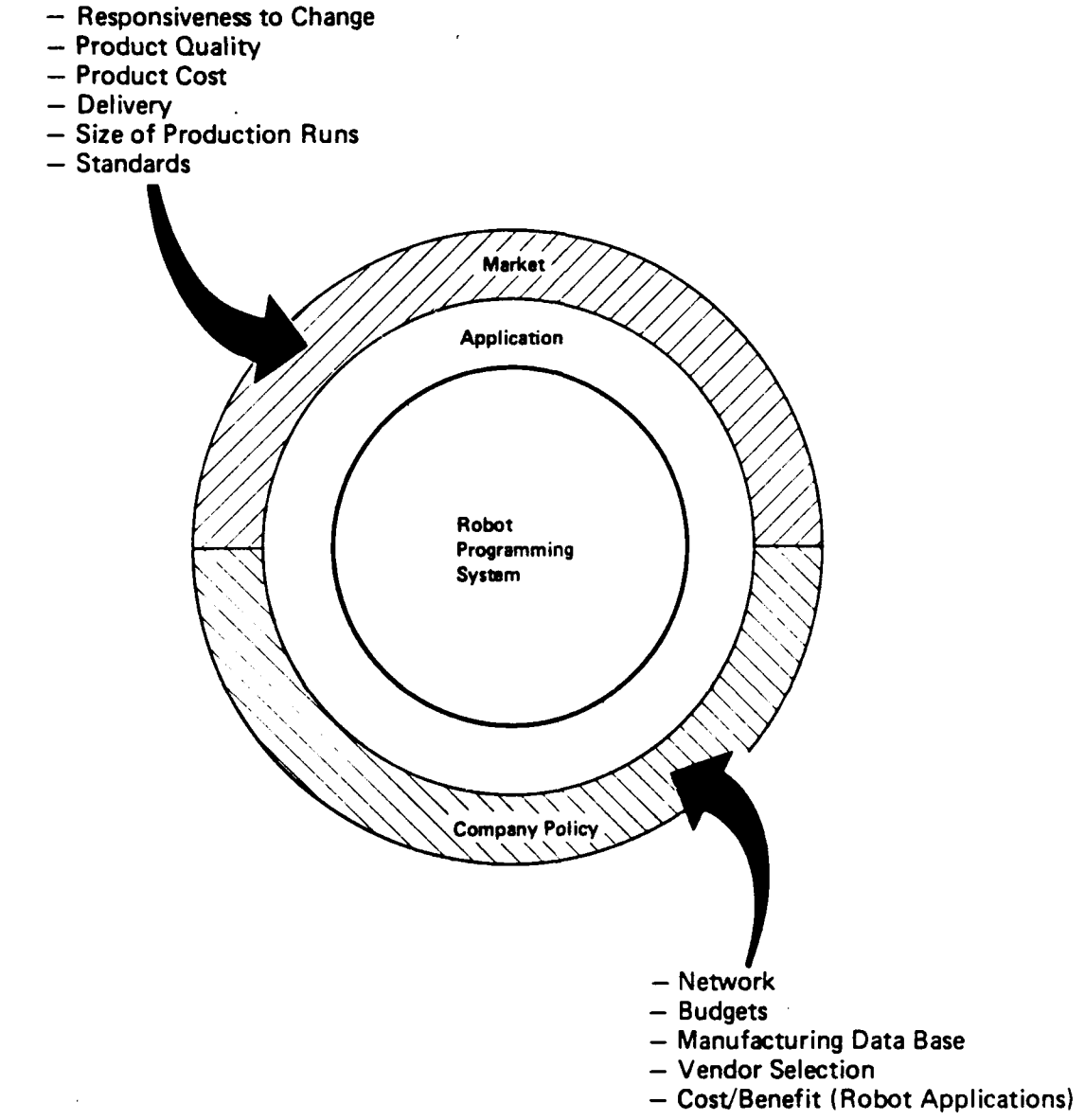


Figure 5. Impact of the external world on the robot programming system. (Source: IITRI).

Responsiveness to change implies the ability of a company to respond to changing market conditions through the introduction or modification of a product line. The programming environment and implementation layers may have to satisfy this demand by being flexible enough to allow frequent reprogramming. Additionally, the language features may have to support significant sensor integration to reduce the need for expensive special tooling for each product.

Depending on the product quality requirements, sensors may need to be used to position the robot within close tolerances. The use of sensors to aid in positioning a robot may allow less accurate robots and more general purpose programs to be used for an application, with a corresponding reduction in product cost.

Production and delivery schedules place great demands on the programming environment and implementation layers. Small and mid-size production batches may not be able to justify lengthy program development cycles, particularly for one-time-only products. On the other hand, industries catering to the defense market sometimes receive an order for a batch size of one, spaced at irregular intervals of a year or more. Of prime concern is the ability to maintain a program already developed for the application, perhaps incorporating newly established process parameters, tolerances, or equipment, without major reprogramming effort. The use of sensors for robot positioning is one way to enhance the reusability of a program for slightly different parts or tolerances.

Should market standards be developed for a product line, all of the layers will be affected to some degree, either through more rigidly defined manufacturing tolerances or by imposing restrictions on the design, communication and use of manufacturing software.

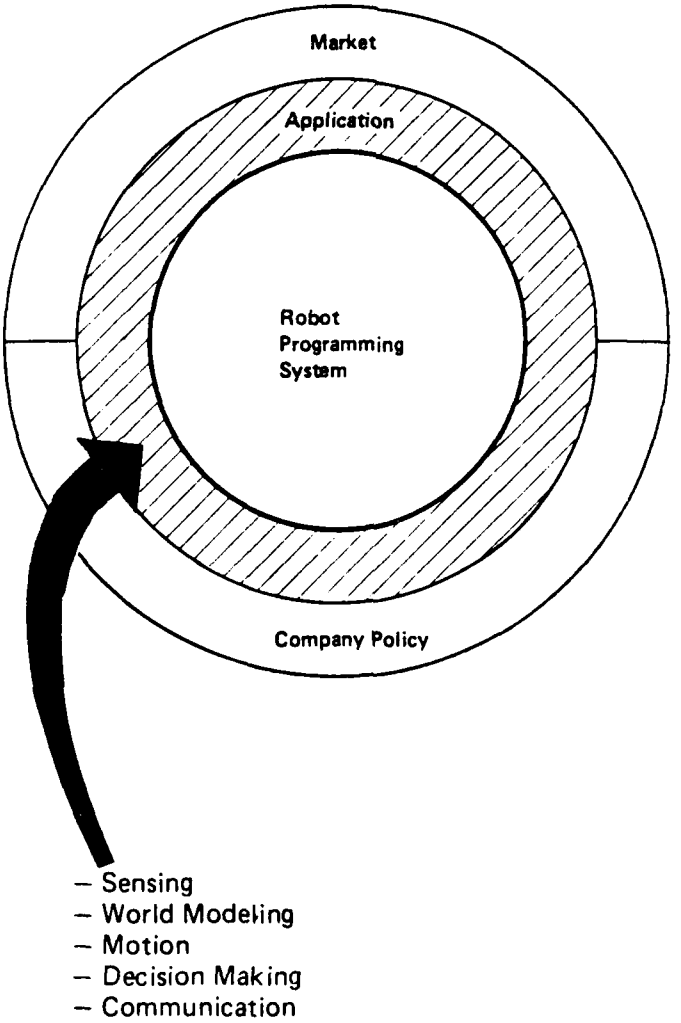
In a similar manner, the entities involved in company policy affect the choice of a robot programming system. If a company is committed to integrated manufacturing using a factorywide communication network, such as Manufacturing and Automation Protocol (MAP), robot controller hardware and software will have to be MAP compatible. Budgetary considerations may affect the sophistication of the programming environment. CAD/Graphics packages may boost programmer productivity, but can be very expensive. The existence of a manufacturing data base may require that the robot language access CAD models. A

company may have committed itself to a particular robot vendor's equipment, which significantly narrows the choice of a robot language. Finally, cost benefit analyses will determine the applications for which a robot will be used. This in turn influences the language features required. While this is only a conceptual treatment, it illustrates the point that company policy factors are a significant concern when evaluating robot programming languages.

3.2.2 The Application Layer

When one begins to compare robot languages, it is important to consider the application. In a palletizing operation, macro capability is obviously important. In an assembly operation, guarded and compliant motion is important. In a sorting operation, vision system integration is important. Very little has been published regarding what language features are important for different applications. Hanright (27) lists some software considerations for arc welding: coordinate system transformations, tool center point specification, linear and circular interpolation, offset displacement, and three-dimensional coordinate shifts. Similar publications would be a valuable aid to evaluating languages, but they seem to be lacking in the current literature. If the user is selecting a robot for only one particular application, a powerful language may not be required. If, on the other hand, the applications will vary or will be complex, it is important to select a robot with a programming system that provides the necessary power and flexibility.

Figure 6 illustrates the entities at the application layer. This layer has been developed to a great extent in the literature. Lozano-Perez (28) presents an excellent discussion of the requirements that an assembly application puts on the robot language. He describes an assembly task as follows: (1) a robot picks up a randomly oriented pump cover from a moving conveyor belt and places it on a base, while (2) a second robot picks up pins from a second conveyor belt and inserts the pins through holes in the cover and base. The system incorporates two robots, a vision system, force sensing, a parts feeder, and two conveyor belts. By reviewing the operation of the system, Lozano-Perez illustrates a majority of the requirements related to robot programming. He summarizes the resulting requirements into five major aspects: sensing, world modeling, motion specification, flow of control, and programming support. Craig (29) and others summarize robot programming requirements into very similar categories.



86677RK

Figure 6. Entities of the application layer. (Source: IITRI).

We base our analysis on the categories set forth by Lozano-Perez. The last category, programming support, is considered part of the programming environment, and is treated in Section 3.2.4. Also, "flow of control" has been renamed "decision making" and interpreted with a slightly broader scope. Finally, an entity called "communication" has been added. Lozano-Perez discusses this under "sensing" and "flow of control," but here it is treated explicitly due to its importance.

3.2.2.1 Sensing

Early robots had very little sensing capacity. Fixtures were designed to maintain a very structured environment so that everything could be predetermined at programming time. In many cases the fixturing costs were equal to or exceeded the robot system cost, but this could be justified in large production runs. Sensors have been integrated into robot systems over the years. This is a result of decreasing sensor costs and increasing complexity of robot applications. Also, special fixturing cannot be cost-justified on small batch jobs, so sensors must be used to compensate for a less structured environment.

Lozano-Perez describes four uses of sensors in robotics. The first is to initiate and terminate motions. For example, to synchronize robot actions with other equipment, a switch closure may indicate that some parallel task has been accomplished and that the robot can continue with its program. The second use is to choose among alternatives. After the robot has attempted to pick up a part, for example, it may check the gripper to determine if it has really achieved its objective. If it has it will continue. If not, it may attempt to pick up the part again, or it may execute some other portion of the program. The third use is identification. A vision system may be used to identify a part and its orientation on a conveyor prior to pick up. The fourth use is compliance. This is executing a function while continuously operating under a constraint. When inserting a pin in a hole, for example, the entire operation must be performed without ever exceeding a specified force. The force sensor must be read continuously during the operation. In general, sensing involves reading sensor input (either continuously or at intervals), manipulating the data, and performing some function based on the results.

As languages evolve toward the task level, sensors will become increasingly important. The task "put box A on box B" requires vision to determine the location and orientation of the boxes, and to provide information so that an appropriate path can be generated. Force sensing will also be required to determine the force necessary to pick up the box without it slipping, and perhaps sensing to determine if it has slipped. This is in contrast to the manipulator level, where the location, orientation, and forces all have to be predetermined at programming time. The only sensing required at this level may be to determine slippage. The application determines the entities required in the various layers of the robot programming system. For example, in the capabilities layer, sensor support is required. In the programming environment, simulation of sensor input is helpful (note that this is not currently available). In the operating environment, interrupts are important. In the feature layer, a number of features including I/O, variables, and arithmetic operations are required.

3.2.2.2 World Modeling

World modeling is the method for describing the environment in which the robot operates. The environment includes objects to be manipulated by the robot, as well as other objects within the robot workspace (machine tools, conveyors, fixtures). For the most part, this is done with coordinate frames which are attached to each object (Figure 7). The relationship between objects is then represented by the relationship between their coordinate frames, a relationship which can be defined mathematically by a coordinate system transformation function. Some languages allow two coordinate frames to be "affixed" to each other. Once a pin has been inserted in a hole in a plate, a language statement allows the pin and plate to be affixed. When the plate is moved in a subsequent operation, the location of the pin is automatically updated.

As languages evolve toward task level, modeling will become crucial. Sensor information will be transformed into world models that inform the robot controller of the current state of the workcell. The model will include location and orientation of objects, as well as activation signals and the state of other communication processes. Frames will be insufficient in a true task level language. Models that depict surfaces and part features and possibly other data, such as center of gravity and coefficient of friction, will be required. Consider the "put box A on box B" example. The vision system will

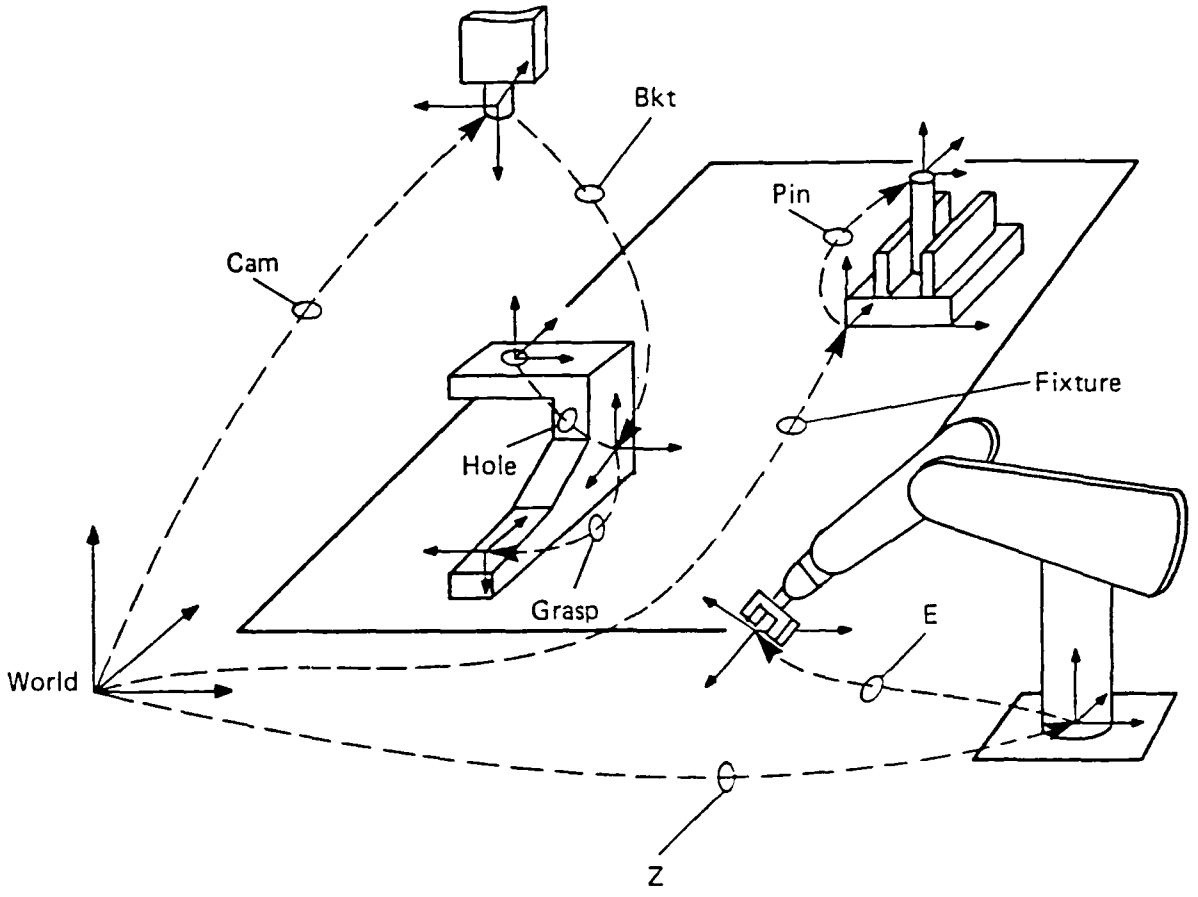


Figure 7. World modeling using frames. (Source: Lozano-Perez, Tomas, "Robot Programming", Proceedings of the IEEE, Vol. 71, No. 7, July 1983, P. 825).

provide information about box A and any objects in proximity. From this information (world model), the robot must calculate a grasping strategy--a method for approaching the box without colliding with surrounding objects. Center of gravity information may be important in determining where to grasp the box. Coefficient of friction information may be required to determine the appropriate grasping force. Finally, the world model will have to be continuously updated as the robot moves toward box B so that a collision-free path can be ensured.

The application layer determines the impact that modeling has on the other layers of the robot programming system. A pick and place operation with special fixturing may require no modeling capabilities while an assembly operation may require extensive modeling. In the programming environment, reach testing, collision detection, and cycle time analysis are all affected by the extent of modeling capabilities. Language features required to support modeling include frames, transforms, and geometric data types.

3.2.2.3 Motion

Obviously an important entity in the application layer, the desired path type imposes requirements on both the programming environment and language features. In some cases, the exact path taken by the robot is not critical, so motion may be specified by the endpoints. In other cases, the path may need to be specified. Many languages allow a linear or circular path to be specified. Speed, and sometimes acceleration and decceleration, are other important parameters in motion specification.

As languages evolve toward the task level, the ability to define complex paths will become less important. At the manipulator level, especially when sensors are not integrated, an exact path must often be specified to ensure no collisions. At the task level, it may be that no motion specification at all is required. The robot will instantiate required motion from the specified goal, i.e., "put box A on box B." Even at the task level, however, it is likely that motion statements will be included as a specification of target points: "go to point X," "pick up box A," "go to point Y," "put box A on box B." This simple task decomposition done by the programmer can take a great computational burden off the robot--it would not have to determine approach

points for box A and box B. Also, there may be times when the programmer does not want the robot to generate its own path, and therefore requires explicit control over the robot motion.

Motion requirements at the application layer impose various requirements in the language features. These include position, path, move statements, coordinate frames, compliance, and others.

3.2.2.4 Decision Making

This involves the ability of the robot to respond to sensor input and take the appropriate actions. It impacts all layers of the robot programming system.

In manipulator level languages, decision making must be explicitly programmed. A sensor input is read, the data is manipulated, and the program branches based on the result. Any action in response to sensor input must therefore be pre-programmed. For example, in "putting box A on box B," a manipulator level program must explicitly test a gripper sensor after attempting to pick up box A. If the box has not been picked up, the programmer will have to have written a recovery program which the robot can then execute. The programmer must try to foresee any problem and specify the action to be taken in its event. It is for this reason that as much as 80% or more of a manipulator program can be devoted to exception handling. At the task level, decision making will still be very important and complex. It will rely heavily on the world modeling capabilities to try and determine an appropriate response to an error condition. Even at the task level, it is likely that the programmer will play a large part in determining appropriate responses. If the problem is not constrained, the robot is faced with an infinite number of possible errors and responses. Consider "put box A on box B" and assume that a box C sits on box B before the task is initiated. If the problem is not constrained, the robot must determine what to do with box C. Obviously it should be removed, but where should it be put? Does it place it somewhere temporarily and move it later? Does it inform some other process that box C is there and then wait for instructions? Should it just place box B on top of box C? Decision making strategy is a complex problem that will grow in magnitude as languages evolve toward task level. Artificial intelligence techniques are likely to play a significant role in developing solutions. Some ongoing research is cited in Section 5.

Decision making at the application layer impacts all layers of the robot programming system. Interrupts in the operating environment and conditionals in the language feature layer are examples.

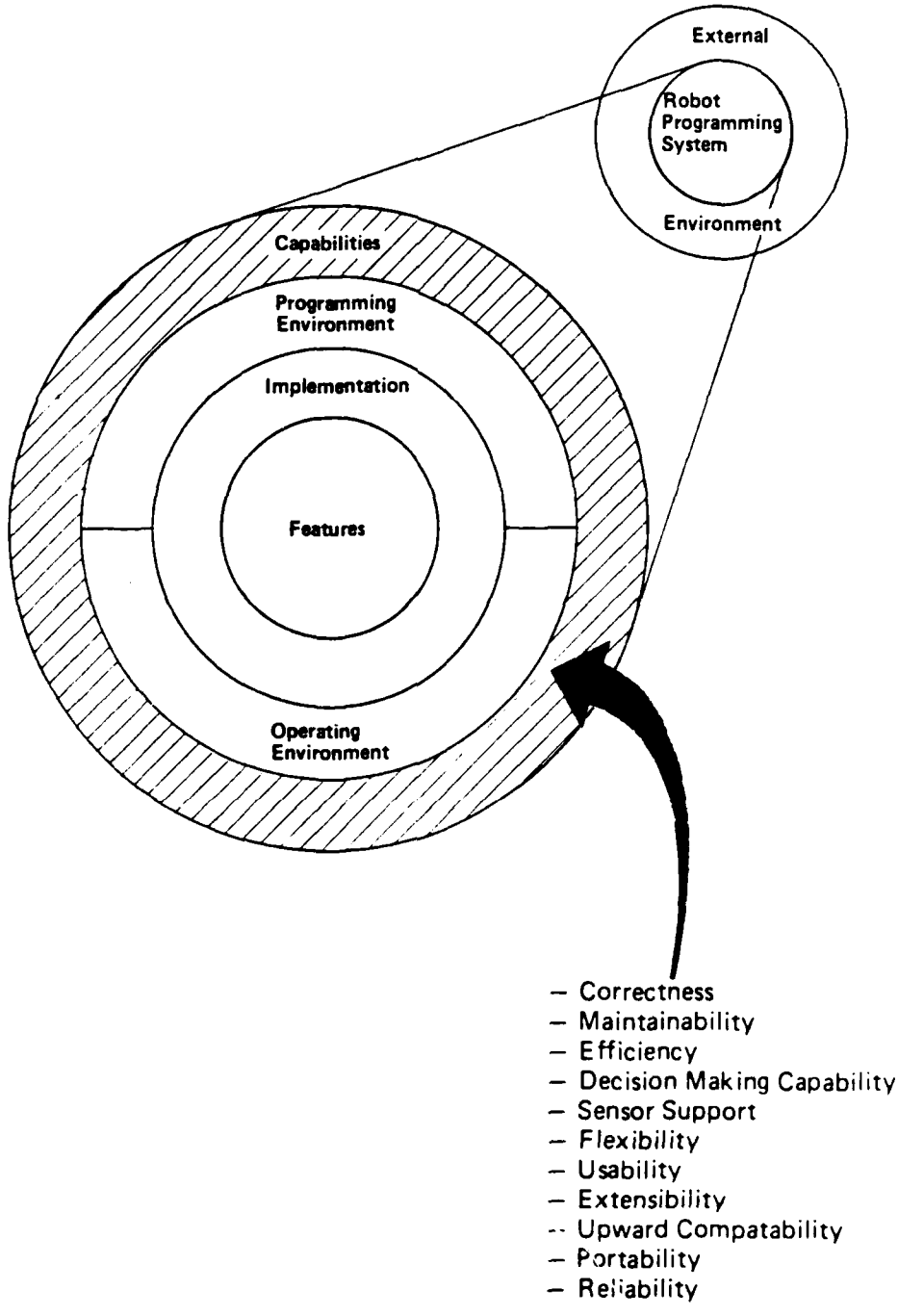
3.2.2.5 Communication

Communication involves both file transfer and synchronization of robot operation with external equipment. At the manipulator level, synchronization typically involves binary signals (switch closures) that alert the robot to the condition of external equipment. Also, some languages allow control of multiple robot arms under the same program by providing some multitasking language constructs. Manipulator level languages have begun to incorporate networking capabilities so that communication with other computers (vision systems, cell host computers) is possible. File transfers, so that robot programs can be downloaded from a host, are also supported. Robot controllers are being built with MAP capability, in recognition of the market. As languages progress toward task level, communication will become extremely important. Transfer of information to and from the robot, probably at the symbolic level, will be essential to updating world models and coordinating robot operations with other equipment. In complex assembly tasks, multiple robots executing programs in parallel will be required. The "put box A on box B" problem requires a significant amount of information processing and information transfer.

Communication requirements affect the operating environment and the feature layer. In the operating environment, networking and peripheral support are required. In the feature layer, sensor I/O, file I/O, device support, and other features are required.

3.2.3 Language Capability Layer

Language capabilities (the "ilities") are discussed in Section 3.1.2. These are entities that describe characteristics of the programming system. They impact the layers beneath (programming environment, operating environment, language implementation, and language features), and are affected by the layers above (application, industrial environment). The language capabilities we have chosen are shown in Figure 8, and are discussed briefly below.



8668IRK

Figure 8. Entities in the language capabilities layer. (Source: IITRI).

Flexibility. The flexibility of a language is a function of syntactic issues, but, more importantly, is a function of the semantic power available to the programmer to represent his application. It is a measure of the range of applications for which a language can be used.

Usability. A language is usable for an application if program development meets acceptable guidelines with respect to cost and maintainability.

Extensibility. The language extensibility is based on the ability to create abstract data structures representing elements of the problem. Without data abstraction capability, hierarchical decomposition of the tasks and modular development are hampered.

Upward Compatibility. Revisions to a language should allow for old programs written in earlier versions of the same language to run successfully.

Portability. The more removed a robot language is from an application area and controller hardware, the more portable it is. Thus, conventional languages such as FORTRAN, C, and PASCAL can be implemented on a wide variety of computer architectures. Portability can be at odds with the programmability, in that the naturalness of the language to express the problem may depend on language features specific to the application area or hardware.

Reliability. This issue is tied in with the mechanics of program development and implementation. Compiled programs are in some ways more reliable, since syntactic and semantic errors can potentially be screened out prior to execution by the controller. The reliability of a program is also a function of the depth to which alternative runtime conditions are handled.

Correctness. The issue of correctness for robot programs is defined simply as "Does it work?" No amount of off-line checking can assure that the program will successfully complete its task.

Maintainability. Easily written and read programs are more maintainable than low-level programs which tend to hide the semantics of the program. Modular development and documentation aids increase the programmer's ability to quickly and correctly modify existing programs as necessary.

Efficiency. Languages which can concisely represent frequently used robotic functions generally achieve faster execution speeds.

Decision Making. Conditional branching facilities of a language are used fully in robotic applications, due to the nondeterministic nature of physical interaction with the real world. Some decisions are planned choice points in a program, perhaps based on sensor input. Others are implicit exception handlers designed to handle classes of unexpected events.

Sensor Support. The ability of a language to support a wide range of complex sensors increases its flexibility and decision making capability.

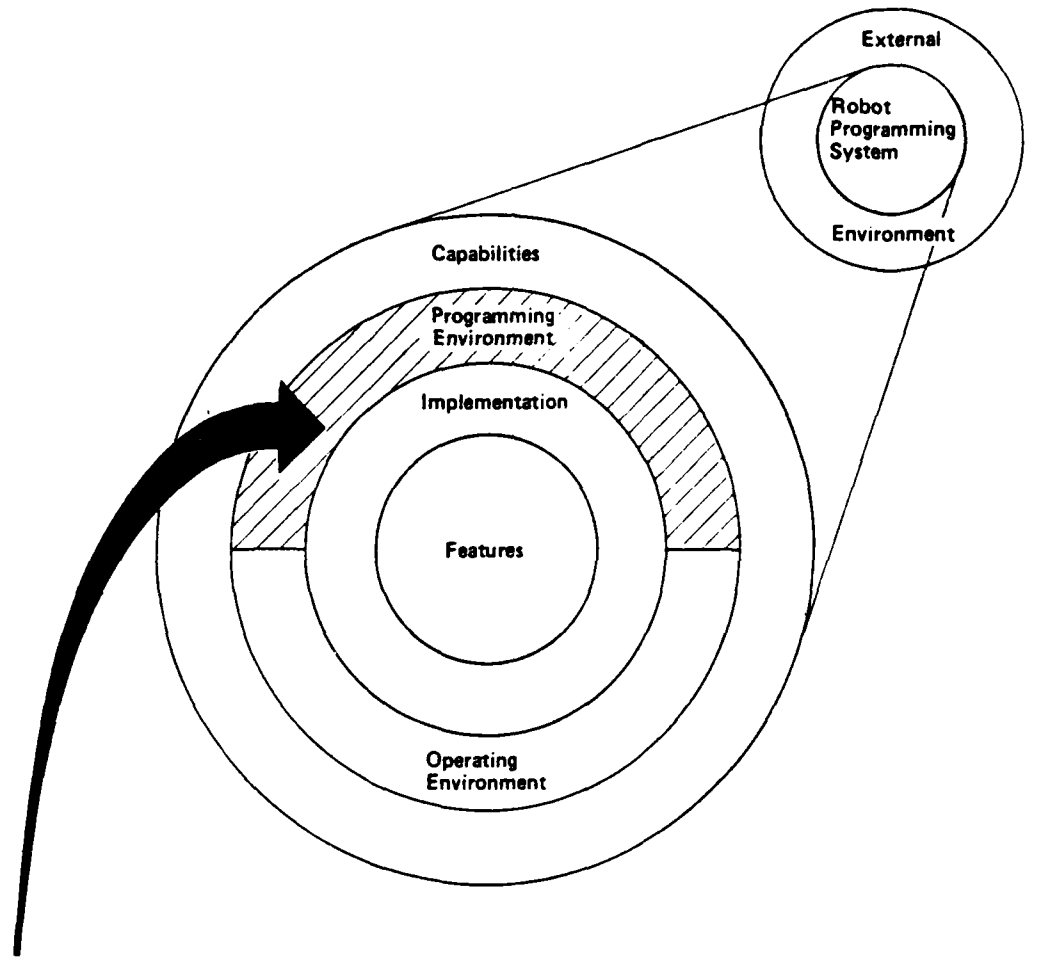
3.2.4 Programming Environment Layer

The programming environment (Figure 9) is the hardware and software with which the programmer generates a robot program. A text editor, a CAD/Graphics system, and a subroutine library are all part of the programming environment. One of the issues in the debate over robot programming techniques is the level and type of programmer experience required. Programmer experience is also an entity in the programming environment. This layer is heavily impacted by the implementation of the language. As discussed in Section 1, interpreted languages are more easily debugged than compiled ones, but are slower to execute. Consequently, some robot programming languages are interpreted, some are compiled, and some are both. In this last case, the program is debugged using the interpreter and, when ready to run, is compiled for efficiency. For these and other reasons, the implementation layer somewhat overlaps the environment layers. Entities in the programming environment that ultimately affect the robot programming system are described below.

Programming Techniques. Programming techniques are discussed at length in Section 1.5. These are obviously part of the programming environment.

Programmer Experience. The methods, patterns and knowledge (hence experience) which the robot programmer can bring to bear on the task significantly impact the effectiveness of the programming tools and techniques available. Additionally, the learning of a new language is greatly enhanced by the range of languages already known to the programmer and by formal training in languages and computer architecture.

Pre-processor. A pre-processor takes in a program having syntax extended beyond the robot language, which it then converts to the normal output language suitable for the remainder of the translation stages. An applications



- Programming Techniques
- Programmer Experience
- Preprocessor
- Editor
- Off-Line Debugging
- Simulation
  - Reach Testing
  - Collision Detection
  - Cycle Time Analysis
- CAD Data Base
- Library
- *Documentation*

86680RK

Figure 9. Programming environment entities. (Source: IITRI).

program written on a CAD/Graphics station is pre-processed into a manipulator level robot program, so that the normal translator or interpreter may be used for execution.

Editor. The editor is the interactive tool whereby the programmer creates programs. In the usual sense, this involves just typing in a program, but it need not be so restricted. Most editors will allow the insertion of other files at desired locations and the ability to keep a log of changes made to an existing program. This latter feature is important to debugging and maintainability. Editors do not often "know" the syntactic rules of a language, and thus allow the user plenty of freedom to make typing mistakes or to specify improper expressions. It is well known that "smart" editors, those that will not allow syntactic garbage to be typed in, significantly boost programmer productivity. Grown out of the artificial intelligence community, these editing techniques can be extended even further to allow "templates" of a typical construct to be used by a programmer, as a drafter would use a geometric template to save time and energy. For example, a tool change sequence template, which would have in it all commands generic to this operation, could be invoked whenever needed leaving space only for the programmer to specify which tools are being changed. Additionally, the editor might have built into it rules that disallow certain sequences of harmful or inefficient program statements, such as two successive tool changes. The editor should, in short, assist the programmer, by helping to write the program, and by serving as a focal point for the integration of all of the elements of the programming environment.

Off-line Debugging. At the minimum, a program written offline should be checked for valid syntax before downloading to the robot or workcell controller. If a compiler is used before downloading, this check comes free as part of the translation process. Practically, much more in the way of debugging must be performed before attempting the use of a robot program, and this is the reason for simulation packages.

Simulation. The simulation of the expected robot behavior as specified by a program is an essential ingredient of the off-line development cycle. A simulator should take in a robot program and, at the very least, show the manipulator motions that would occur if actually run on the controller. It is desirable to have as many elements of the environment simulated as possible,

such as "axis windup," load kinematics, process tolerances, sensor delays, and compliance. Each additional model representing these elements burdens the simulator package, but significantly improves the development cycle of complex robot programs. Workcell simulation performs the analysis of data and material flow within the cell and may use task planning (10), cycle time estimation, and sensor models to optimize the synchronization of processes and layout of the cell. Current simulation tools are limited in their power, and improvements are to be expected over time.

Reach testing is a useful component of a simulator. It is the ability to determine if a manipulator can be configured so as to reach a desired location in the workspace and maintain a specified orientation.

Collision detection is critical to the development of workable programs. Without it, the integrity of the manipulator, fixture, tool and part is in serious jeopardy during program testing. Analytically, joint angle, Cartesian coordinate, and spatial representations of the manipulator and environment are used by algorithms to infer possible collisions. Most current systems provide only a visual simulation, and the operator must determine collisions manually.

Cycle time analysis is the ability of a simulator to estimate the total time required to execute a motion sequence, and can be important to estimate cost in large production runs. Acceleration and deceleration factors should be taken into account, as well as gripper activation and roll, insertion and removal times, and part feeder location and presentation (9). Unfortunately, current systems assume instantaneous accelerations and decelerations, and zero settling time.

CAD Database. The information necessary to model a part (but rarely an assembly), for purposes of simulation, is sometimes found in a CAD data base. Depending on the CAD representation, which is often two-dimensional and lacking tolerance data, a conversion may be necessary to produce a part model adequate for the needs of the simulator. If available, the CAD information may also be used in the development of the original program itself. An editor that extracted relevant CAD data for the part being programmed would be valuable. Practically speaking, a CAD data base only has utility in the initial development cycle, although the whole programming environment might reside on a CAD system.

Library. A subroutine library is an indexed file of subroutines or subprograms that have been previously developed to solve frequently occurring programming problems. If the language implementation supports the access of a subroutine library at the time of compilation, the necessary routines are automatically integrated into the output file. An extensive subroutine library will improve programmer productivity, since previous programming effort is reused, and this will also improve program reliability and maintenance.

Documentation. An essential ingredient in all of the language capabilities, some high-level languages are termed "self-documenting" in that the syntax of the language is so natural and readable that the meaning of programs written with it is "clear." This does not imply that further documentation is not important. The development costs of robotic applications demand strict attention to documentation at every phase.

3.2.5 Operating Environment Layer

The operating environment (Figure 10) coexists at the same layer as the programming environment. When taken together they become the actual implementation of the robot system. The operating environment consists of the robot, the controller, sensors, any external machines and conveyors, other computers, and operator terminals. Robot programming system entities include communication with the operator, runtime debugging, and error logs. They support execution of the robot program, and are discussed below.

Execution Sequence. A robot program is composed of a number of statements whose execution (evaluation) may require very specific scheduling to achieve even reasonably correct control over the robot and the supporting environment. The scheduling is under partial control of the program (see concurrent execution), but is often provided by the specific implementation (see multitasking) and depends on the underlying computer hardware architecture (see parallel processing). This hidden scheduler is a large source of runtime exceptions and an unknown quantity as far as off-line simulation and debugging are concerned.

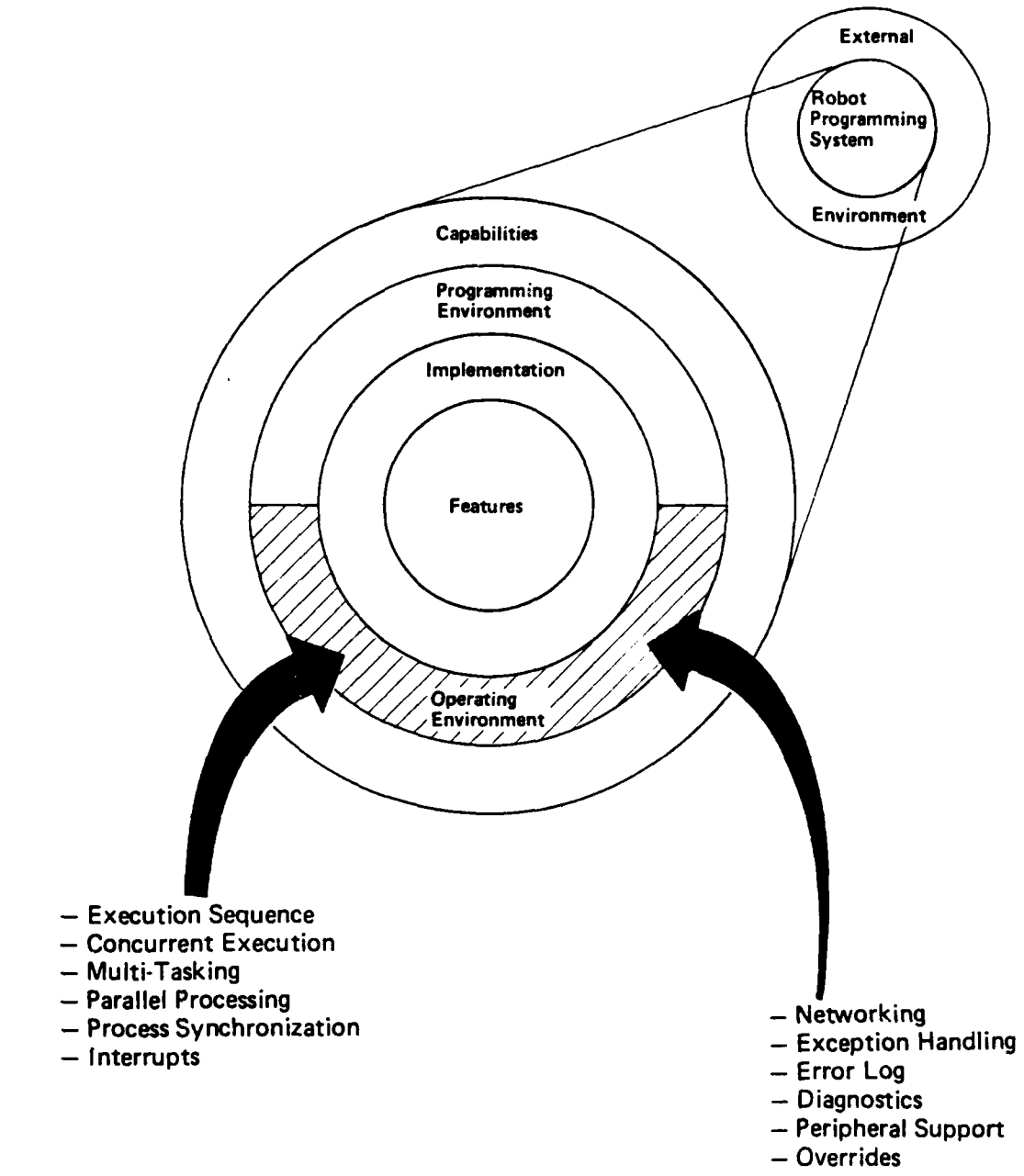


Figure 10. Operating environment entities. (Source: IITRI).

Concurrent Execution. When two or more program activities (such as statements or procedures) are to be performed simultaneously, the programmer may specify this using a "COBEGIN...COEND" or "PARBEGIN...PAREND" language construct. The ability of the controller having this language to perform simultaneous execution in this manner is highly dependent on the underlying computer architecture. The program activities to be performed in this manner are temporarily given the resources of separate microprocessors to accomplish this. A common usage in robotics is to acquire images from a camera while moving the manipulator.

Multitasking. Some computer architectures have enough power and flexibility that they can have more than one program at a time in some state of execution. For example, while "Program A" is idle, or waiting for some event like a switch closure, "Program B" can be initializing the robot position. Both programs are in a state of execution, yet only one of them has control at the moment. The scheduling mechanism that swaps control from one program to another is performing multitasking, and is different than parallel processing.

Parallel Processing. Some computer architectures allow for several actual processing units to access an area of memory common to them all. These microprocessors are performing their own functions, somewhat independently of one another. This parallel processing is common to robot controllers, but is by no means easy to build or control. Typically at least two threads of parallelism are in controllers - the servo computers and the top level program evaluator. Some controllers have additional layers of control acting in parallel, such as trajectory computers, I/O handlers, and more.

Process Synchronization. The orchestration of control may be explicitly specified, up to a point, in a single program itself. On the other hand, several concurrent processes may be controlled if a synchronization mechanism is available between the individual programs. Systems allowing multitasking usually provide a channel of communication between concurrently executing programs, which enable one program to "wait for" or "signal" another. An example of this may be found in an integrated vision system, where the manipulator program and the vision program might be running continuously, but the manipulator program has to "wait for" an image to be acquired and processed before reading the results.

Interrupts. The ability of the robot program to examine the I/O ports to see if information is there, is termed polling. On the other hand, the arrival of information at a port may interrupt a running program, thereby forcing it to take notice. The first case is exemplified by the language construct "WAIT FOR EVENT," which continuously scans (polls) its ports for the arrival of information (an event), such as a switch closure on a fixture. Interrupts, however, often signal the program that an unplanned event that needs immediate attention has occurred. An example of this is the closure of a travel limit switch on a manipulator joint.

Networking. A number of schemes exist whereby a controller may share resources, such as files, with a computer located in another area of the plant. A network includes the data transmission cables running between the "nodes," the special purpose hardware in each node, and the programs that serve to process the data transmission. A number of networks exist. Some have grown out of general purpose computing, such as "Ethernet," while others like MAP were developed with automation in mind. Networking tends to vastly improve the effectiveness of robotics when substantial information pertaining to the process is available on some *in-house computer (i.e., a cell host* computer). In use, the network will be transparent to the robot programming language, which is to say that the program does not need to know where the files are; the network level program handles that.

Exception Handling. Runtime errors are those conditions arising from hardware failures, unanticipated for sensor data, or misuse of a control mode. In the case of program development which does not include compilers and off-line debugging, syntax errors may also cause runtime exceptions. Exceptions resulting from unplanned events may sometimes be handled via "condition handlers." These error trap routines are designed to take control when some pre-defined condition occurs. Within them, some corrective action is taken if possible, and the program is resumed. A great deal of robot programming is concerned with detecting exceptions and processing them. Until recently, such errors would cause a "hard failure"-- a halt condition. Newer languages allow "softer" degradation, which is an important safety issue.

Error Log. The occurrence of a runtime error may cause a notation to be placed in a file for analysis by the programmer. Useful information, such as system variables, the state of I/O, program line number, and the time, should be placed in the file as well. Without this information, the causal determination of an exception may be hampered.

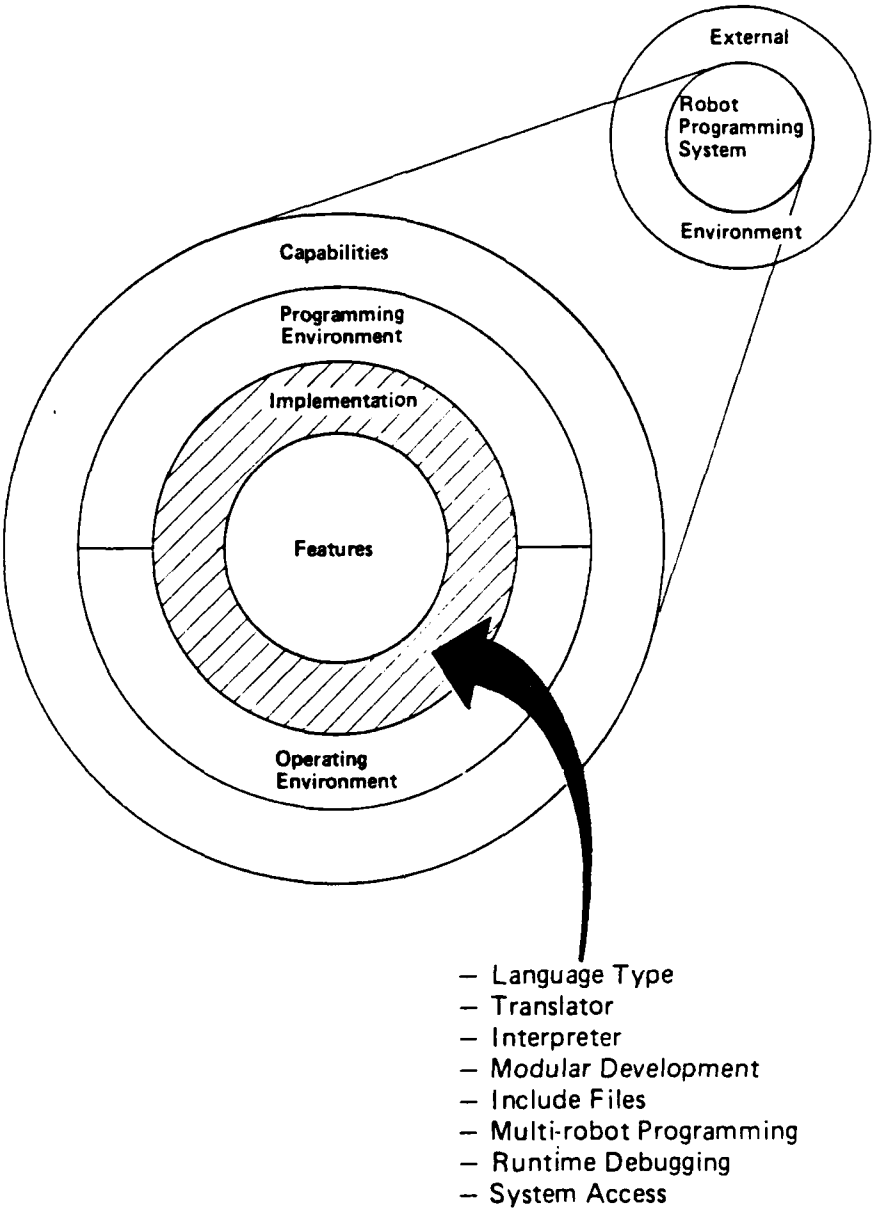
Diagnostics. A controller needs a set of very low-level programs, possibly hard wired into the computer architecture, that test and evaluate system components. They are often invoked automatically on powerup, but may be run as desired to check the integrity of memory, communication, and lines. As a common illustration, personal computer users are subjected to a delay on powerup while such programs are being run automatically. This should not be confused with debugging.

Peripheral Support. A language supports I/O in a generic sense to logical devices. At the controller level, however, low-level programs which accomplish the transmission of data through ports to conventional data processing peripherals as well. Printers, disk drives, display screens, and keyboards fall into this category. Peripheral support of this type is wholly the responsibility of lower-level controller specific software.

Overrides. Hardware overrides enable the person operating the controller to change modes of operation from teach to edit to run, etc. Also available are maximum speed controls, limiting the actual motion speeds of a program.

3.2.6 Language Implementation Layer

The implementation of a language (Figure 11) is depicted as the layer in the robot programming system residing between the operating and programming environments and the innermost feature layer. The key issues involved here are represented as entities for the type of language, how the language is processed, and how the actual configuration enables certain kinds of debugging and system access. The implementation of a language is the view of the language which is seen by the outer levels. It serves to incorporate the designed language features into a workable system, to enable use by the programmer and robot controller.



86678RK

Figure 11. Implementation layer entities. (Source: IITRI).

Language Type. Programming languages are classified according to high-level, distinctive syntactic and semantic features. Languages such as PASCAL, C and KAREL are structured, while BASIC, FORTRAN and VAL are unstructured. Languages may also be oriented in different ways. KAREL is procedurally oriented like many computer programming languages, while AML/X and LISP are expression oriented. Object-oriented languages, such as SMALL TALK, have had a significant effect upon robot task level programming approaches. Goal driven languages, an example of which is PROLOG, will play an increasing role in task level languages. Finally, a language is implemented as an interpreter, compiler or some combination of these.

Translator. A translator is a program that reads in "source language," perhaps a robot program, and then outputs a lower level "object language." This generic concept is the basic action of compilers and pre- and post-processors.

A compiler reads in higher order language and puts out a language similar to the language actually used by the computer architecture running the program, program, i.e., the target computer/controller. A compiler runs in the batch mode--an entire program is read and compiled, and then the object code is output.

Interpreter. An interpreter is a program that runs on the target controller and reads in source (high) level program statements one by one and executes them directly. Off-line translation is avoided, but with some penalties. Syntactic errors in the source program may appear as runtime exceptions. Also, the interpretation of a program statement requires significant overhead before it can be executed. This is why interpreters tend to run slower in the real world than one would like, particularly since typical control structures cannot be optimized as they would be by a compiler. The greatest benefits of the languages that can be run with an interpreter are their power and ease of debugging. Much of the power comes from the notion of delayed binding and dynamic scoping.

Modular Development. Larger programs benefit from partitioning the job into smaller pieces that may be written somewhat independently. These program modules can be individually translated and perhaps simulated before the complete program is assembled. Incremental compilation supported by some languages allows for modular development.

A well designed programming environment will promote the use of one module, perhaps with a few minor changes, for a variety of applications. This issue of reusability is borne of software engineering practice that claims it to be a major contributer to productivity in the future of all programming.

Include Files. During compilation or pre-processing, some language implementations will allow for the automatic insertion of designated program or data files into indicated areas of a source program. Include files differ from subroutine libraries in that a library contains previously compiled and tested program segments, while a file to be included might be untested source code or data.

Multi-Robot Programming. Coordinating the control of more than one manipulator on a given task is an extremely difficult problem. Complications arise from the extra degrees of freedom present, timing constraints, and the need to (perhaps) apply forces upon another manipulator through simultaneous contact with the same rigid body. Motion simulation of multiple robots is currently possible, but more complex models incorporating forces and sensors will be needed to simulate intricate assembly tasks.

Runtime Debugging. A program under development is eventually executed on the controller, in as realistic a fashion as feasible. This is where the remaining semantic and process synchronization errors are discovered after the syntactically correct program has been downloaded. Tools are available to assist the programmer in tracing, breaking or otherwise monitoring the program execution. Depending on the level of language actually executed at the controller, i.e., man- or machine-readable, the program may be modified with a runtime editor.

The controller may show the program statement currently being executed on its display screen. This is tracing, and is only useful after significant off-line program development.

Another runtime debugging technique, called single stepping, enables the execution of a single program step, followed by a pause. Using this, the robot may be slowly stepped through its motions.

Breaking is similar to the mechanism of tracing, but considerably more powerful. At desirable points during a run with the robot, program execution may be temporarily suspended. At this time, the on-line programmer has the

option of examining the values of variables, the system's registers and ports, and perhaps overriding some program commands. After doing so, the suspended program may be resumed. This is the most sophisticated debugging technique available to the programmer, and has utility only during execution. Although this technique is used within the operating environment, it is provided by the language implementation.

Depending upon the level of the language implemented on a controller, a runtime editor may be used to make minor program modifications. As the controller is not designed for difficult program development, very few tools will be available at this stage of testing.

System Access. If a robot program requires access to hardware or software resources beyond those explicitly provided by elements of the language, the program requests the assistance of the system monitor program. This monitor is often referred to as the operating system or supervisor, and is present to some degree on virtually every computer or sophisticated controller. Access to the system monitor is especially important to programs embedded in a workcell controller, which is orchestrating the communication and control of multiple robots, devices, and sensors.

3.2.7 Language Feature Layer

Much of the literature focuses on the features of different robot programming languages (Figure 12). We have argued that the features are the innermost layer of the programming system, and that they must be put into perspective if an evaluation of the features is to be used as a measure of a programming language. Simply listing the features available in a language does not tell one much about the language, unless these can be used to make conclusions concerning the operating and programming environments, the language capabilities, and the potential applications. In this section the features of a robot programming language are defined. Many of these features are common to conventional computer languages, and some are unique to robot programming.

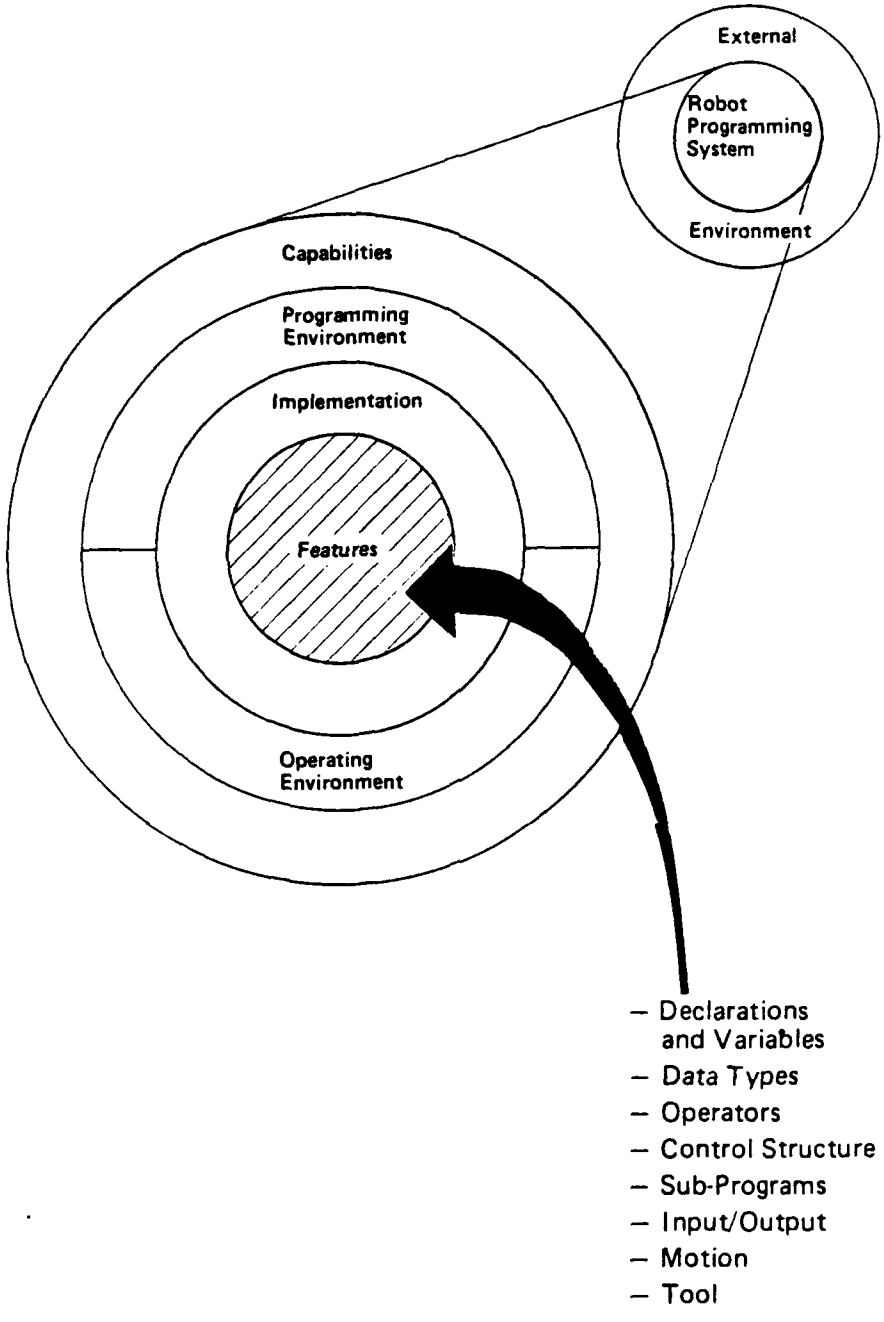


Figure 12. Feature layer entities. (Source: IITRI).

3.2.7.1 Declarations and Variables

Variable Type. All of the variables used in a program have an explicitly stated data type if specified in a declaration, otherwise they have an implicitly assigned type depending on the name or usage. The lack of a full complement of variable types can severely restrict any language.

Variable. A variable is an identifier or name, which may or may not be established in a declaration. If it is, the scope of the variable extends over the program segment in which it is declared. A variable can be given a value explicitly through an assignment statement, or implicitly via parameter passing. Variables have common use throughout all programming. A variable is usually restricted to take on only certain data types or ranges of values, as specified in the declaration. Failure to meet those restrictions may cause exceptions.

Identifier. This is usually defined as a string of characters beginning with a letter which is used for everything from key words denoting syntactic structure to variables and constants. Associated with the latter two is the notion of scope. Systemwide names such as $clock-time or %io-status may exist in some implementations in order to act as identifiers outside the realm of the robot language syntax, for the purpose of referring to precise locations in the controller's memory.

Label. Program statements can sometimes be labelled or tagged, for direct reference. In FORTRAN and BASIC, the labels are integers, while in most structured languages they have the format of identifiers. A labelled statement can have control given to it using a control structure such as a "GOTO."

Constant. A constant is like a variable in most ways, except that its value is established before program execution, and it may not be changed during the execution. The use of a constant name is synonomous with using its value instead.

Declaration. User defined data types, variables, external program modules, subprograms, and labels are specified by name in declaration statements within a program module. They usually preceed the use of the name being declared, and are examined by the compiler prior to the actual execution of the robot program.

Scope. All names used in a program, whether they appear in a declaration statement or not, have defined limits of visibility to portions of the program. This scope determines what meaning is attributed to the name when it is encountered during execution. Static scoping rules are used by a compiler to set up the association between a name and a meaning before execution, while dynamic scoping rules are used by an interpreter and defer the association until needed. Static scoping tends to create optimizable and more reliable execution.

Assignment. During program execution, a variable is given (bound to) a value by an assignment statement. This process may be likened to an operator that evaluates the meaning of the right hand operand(s) and stores it in association with the name of the left hand operand.

3.2.7.2 Data types

Elementary Data Type. This is the most primitive type of data representable in a computer language. It includes arithmetic types (such as integers, reals, complex, double precision, binary), Boolean types (such as true and false), and character types (such as ASCII and character string types).

Structured Data Type. An aggregate composed of elementary data types or other structured data types, which can be manipulated or accessed as a whole or in part. Common examples in conventional languages are records, lists, and arrays. Common to robot languages are frames, matrices, and vectors. These are actually arrays which have special meaning in the robot language.

Geometric Data Type. An instance of a structured data type which is important in robotics, geometric data types specify points, lines, planes, curves, and surfaces. These are needed for referencing part features such as holes.

Array. Arrays are a useful way to aggregate information so that it can be accessed by numerical indices. A common use of an array is to represent a transformation matrix for the computation of one coordinate frame from another. They are also heavily used for two- and three-dimensional maps of an image. A number of elementary data types should be supportable in an array.

Vector. A linear array of numbers, often handled as a matrix, vectors may be used to represent the manipulator's joint angles, Cartesian coordinates, gripper approach information, forces, velocities, etc. There are special vector operations, such as dot product and cross product, which are useful in robotic programming.

Frame. This may be thought of as a specification of a manipulator, tool or object position in space relative to a known origin. The entries of a frame may be given values in response to sensor inputs, initialization, joint encoder readings or calculations based on other frames of reference.

Matrix. This is a rectangular array of numbers. The numbers are called entries, and in robotics they represent positional, orientational and perspective information. There are defined matrix operations, only a few of which find frequent use in robotics. Vectors can be conveniently represented as n x 1 matrices and as such, matrix operations are defined for them as well.

Coordinate System. This is a frame of reference for descriptions of vectors and angles. There are a variety, each having utility to different applications. Common 3D orthogonal ones include Cartesian, cylindrical, and spherical. The specification of orientation can be done with an non-orthogonal system based on three Euler angles or upon roll, pitch, and yaw angles of rotation. There are some standard interpretations for the Cartesian coordinate system - right-handed and left-handed. The right-handed is more frequently used, and can be remembered with the observation that a screw pointed in the positive direction along the Z-axis would be advanced if the positive X-axis is rotated 90 degress toward the positive Y-axis.

Joint Angle. This is an angle of revolution between two prismatic links, joined at a common revolute axis. Joint angles are the positional information in a servo level program.

Origin. In robotics, the origin of a coordinate frame is usually placed conveniently with respect to some manipulator or object feature. For example, the origin of the robot coordinate frame is often placed at the base, but the origin of the coordinate frame for a part might be a protruding corner.

Position. Defines the location of a point in Cartesian or joint angle space.

3.2.7.3 Operators

Arithmetic Operators. These are functions which include addition, subtraction, multiplication, and so on. Also included in this list are trigonometric functions.

Boolean Operators. Boolean operators include NOT, AND, OR, and NOR.

Relational Operators. These yield Boolean values and include equal, less, and greater than.

Transformation Matrix. This is a matrix representing the translation, rotation, and scaling necessary to align one coordinate frame with another. The matrix product of successive transformation matrices allows one to express the position and orientation of a manipulator joint with respect to other coordinate systems.

Frame Affixment. Frames are associated with objects in the robot workspace. A powerful feature provided by some robot languages is to affix one frame to another. When one frame is moved, the other is automatically updated.

3.2.7.4 Control Structures

Branching. The flow of control in a program is determined by a control structure. Branching to other parts of a program may be done on certain conditions using a structure like "IF cond1 THEN goto label1 ELSE goto label2". Unconditional branching is accomplished with a "goto" kind of statement.

Looping. Looping behavior can be created with conditional branching, or with a higher level control structure such as "WHILE cond1 DO this and then this ENDOFWHILE". A variation on this theme is "UNTIL cond1 DO ...."

Iteration. Iteration over a set of entities may be performed with the previously mentioned constructs, or with another such as "FOR each in this set, DO this and then this."

Multiple Arm Control. Industrial robot programs usually provide for the control of only one manipulator. Coordinated motion between two arms has application to sophisticated assembly tasks and cell control. Robot language

constructs to enable multiple arm control under a single program are typically "COBEGIN-COEND" or something similar. For control of multiple arms under separate programs, the language extensions are harder to define and have yet to be developed.

3.2.7.5 Subprograms

Macro. This is an association between one symbolic expression and another. Its utility arises when a larger piece of detailed code can be viewed as a template, with slots that are filled in according to the invocation. For example, if "Increment" is defined as a macro suitably, then the program statement "Increment X" might be expanded to "X = X + 1." An obvious use in robotics is palletizing, where the pattern of motion and control is identical for inserting parts throughout the pallet, but the location of each part is different.

Subroutines. The activities a robot program must perform typically fall into categories, such as motion, sensing, or transformation calculations. Within each of these generic activities are specific actions to perform such as move, rotate, change speed, and grab camera image. Because of the sheer quantity of activities and their differing nature, it is desirable to organize them so that they may be programmed independently from one another. This is a basic neccessity in all of programming, but particularly in a domain as varied and complex as robotics. These independent modules are often called subroutines or procedures. Their liberal use is essential to effective robot programming, but this language construct is not provided by all languages.

Nesting. Nesting is the enveloping of a program construct within another construct. For example, a macro routine can be called from inside another macro. This is a basic capability of structured languages, and leads to better programs.

Parameters. A characteristic of structured, procedural and functional language types, is the ability to pass selected information to and from a subroutine or macro. These data items are loosely referred to as parameters. Larger programs and any that intend to use a given subroutine for more than one specific purpose need the language feature of parameter passing.

3.2.7.6 Input/Output

Binary. Binary (single bit) input and output typically involves switch closures or digital logic signals. This is used extensively for synchronization. Language statements that read binary input are often tests for activation such as "if port1 is on, then goto label2."

Analog. Analog input and output are voltage signals that allow the robot to read sensors and provide controlling output. A force sensor is an analog input. Sometimes specific language constructs are developed to use analog I/O; such is the case with compliance. Otherwise, conditionals are used: "if port1 is less than 5, then goto."

Vision. Obviously important in robotics, special language statements for incorporating vision sensors have been established in robot languages. This is because a vision system normally has its own computer for image processing, and the robot system is concerned only with the results of this computation. Commands such as "adjust," based on vision input, are commonplace.

Compliance. Due to either the nature of the task or to the range of part tolerances, a robot joint may require flexibility in order for the manipulator to successfully complete a task. This lack of stiffness may be built in as with mechanically compliant wrists, or it may be achieved using force, proximity, or vision sensors in a feedback loop. The "peg in hole" task is a common example of the necessity of robots to perform a task while complying with external constraints (maximum allowable force). Compliance requires continuous motion control in response to continuous sensor feedback. For this reason, it is sometimes referred to as "force servoing" or "vision servoing."

Guarded Motion. A guarded move can be thought of as a motion sequence which terminates on condition that an external force is sensed. Some languages provide constructs such as "move until" to perform this function.

Timers. Various hardware timers are accessible from robot program statements of the form "delay 10," "startat 1:05:15." These are necessary for coordinated control of motion, fixtures, and sensors.

Wait. A wait statement continually looks for the occurrence of a desired event, and when (if) it happens, an action is taken. This involves polling of the sensor input, where it is periodically scanned along with other inputs.

Text I/O. Communication between the user and the controller running the language requires some passage of textual data in the form of commands, status messages, error messages, etc. A robot language needs minimally a construct such as "DISPLAY" or "PRINT," and "READ."

File I/O. Depending on the hardware running the robot program, various file system capabilities will be available. A good language will be able to make thorough use of these, for purposes of keeping error logs, or accessing information about a part or process. If the controller is on a network, the files may be stored on another hardware device and referenced from the controller using a file transfer protocol.

3.2.7.7 Motion

Move. Robot languages have been primarily designed to achieve physical manipulation of objects. These actions are typically performed by the execution of primitive motion commands such as "move," "grasp," and "stop" with parameters specifying speeds, acceleration, forces, etc. The extent to which a programmer has control over various move parameters depends upon the language used.

Path. This is the geometric path followed in a move command. Some languages allow paths to be specified in more than one coordinate system. Also, some allow paths to be specified as straight line, circular, or along some other geometric curve.

3.2.7.8 Tool Statements

Effector Command. The end-effector is commonly referred to as the "gripper." Commands issued to the gripper include "grasp," "close with force 3," and "open 6," and are usually expressed in the gripper coordinate frame.

Tool Command. This is the specification of a move relative to the gripper-held tool frame of reference. "DRILL FORWARD 3," for example, might cause a relative motion along the axis of the drill bit, regardless of the orientation of the robot arm. A tool is held by a gripper, and as such it has its own set of operations specific to the tool function. The operation of a tool is accomplished through programmed procedures, which send data out on I/O ports and up to the tool or to the tool controller. The tip of a tool held by a gripper is frequently used as a reference for the positioning of a robot.

3.2.8 Relationship Among Layers in the Programming System

The robot programming system has now been developed to at least a first order level of detail. The entities in each layer have been identified, and some relationships between layers mentioned. Basically, if an entity in an outer layer is considered important, it establishes requirements for entities in one or more of the layers beneath. Ultimately, some language feature(s) must be present in order to support it. It is only within the context of the overall robot programming system that the true value of a programming language can be measured.

A conceptual representation of how the layers of the robot programming system interact is shown in Table 4. Across the top of the figure are the entities in the application layer. Along the side are the entities in layers below. The X's indicate entities which are related to support of the application entity.

A proper approach to establishing the relationships between layers would be to develop a formal data model for the robot programming system. Several formal data modeling techniques exist. The IDEF methodology, for example, was developed by the Air Force and has had widespread use throughout the aerospace industry. Such a model would be extremely valuable for evaluating existing languages and for establishing requirements for next generation languages. With such a model, one could establish the language features required due to various constraints in the different layers. The model would provide a structured analytical framework for use by both users and vendors of robotic software. Having at least conceptually established this framework, we can now properly discuss language comparisons published in the literature.

3.3 COMPARISONS IN THE LITERATURE

There are several good comparisons of robot languages in the literature, but they normally focus on only one element of the programming system. Ahmad (17) compares AML, RCCL, AL, VAL, and SRIL-90 (see Table 5). Most of his comparison is based on language features, but the last few lines of the table are concerned with other layers. Structured language (language implementation layer), operating system (operating environment), and debugging and immediate operations (programming environment) are also mentioned.

TABLE 4. IMPACT OF APPLICATION LAYER ON THE ROBOT PROGRAMMING SYSTEM

		Application Layer				
		Sensing	World Modeling	Motion	Decision Making	Communication
Capabilities	Flexibility	x	x	x	x	x
	Usability	x	x	x	x	x
	Extensibility	x	x	x	x	x
	Upward Compatability	x		x		x
	Portability	x		x		x
	Reliability	x	x	x	x	
	Correctness	x	x	x	x	
	Maintainability	x	x	x	x	x
	Efficiency	x	x	x		
	Decision Making	x	x		x	
	Sensor Support	x	x		x	
Programming Environment	Techniques	x	x	x	x	x
	Experience	x	x	x	x	x
	Editor	x	x	x	x	x
	Off-Line Debugging	x	x	x	x	x
	Simulation	x	x	x	x	x
	CAD Database	x	x	x		
	Documentation	x	x	x	x	x
Operating Environment	Execution Sequence					
	Concurrent Execution	x	x	x		x
	Multi-Tasking	x	x	x		x
	Parallel Processing	x	x	x	x	x
	Process Synchronization	x	x	x	x	x
	Interrupts	x			x	x
	Networking					x
	Exception Handling	x		x	x	x
	Error Log	x		x	x	x
	Diagnostics					
	Peripheral Support					x
	Overrides				x	x
Language Implementation	Language Type	x	x	x	x	x
	Translator	x	x	x	x	x
	Interpreter	x	x	x	x	x
	Modular Development	x	x	x		
	Include Files	x	x	x		x
	Multi-robot Programming	x		x		
	Runtime Debugging	x	x	x	x	x
	System Access	x				x
Language Features	Variables	x	x	x	x	x
	Data Types					
	. Geometric Data	x	x	x		
	. Array	x	x	x		
	. Vector	x	x	x		
	. Frame	x	x	x		
	. Matrix	x	x	x		
	. Coordinate System	x	x	x		
	. Joint Angle	x	x	x		
	. Origin	x	x	x		
	. Position	x	x	x		
	Operators					
	. Arithmetic	x	x		x	
	. Boolean	x	x		x	
	. Transformation Matrix	x	x	x		
	. Frame Affixment	x	x	x		
	Control Structures	x	x	x	x	x
	Sub-Programs	x	x	x	x	x
	Input/Output					
	. Binary I/O	x				x
	. Analog I/O	x				x
	. Vision	x	x			x
	. Compliance	x	x	x		
	. Guarded Motion	x	x	x	x	
	. Timers	x			x	x
	. Wait	x	x	x	x	x
	. Text I/O					x
	. File I/O					x
	Motion Statements	x	x	x	x	x
	Tool Statements	x	x	x	x	x

Source: IIT Research Institute

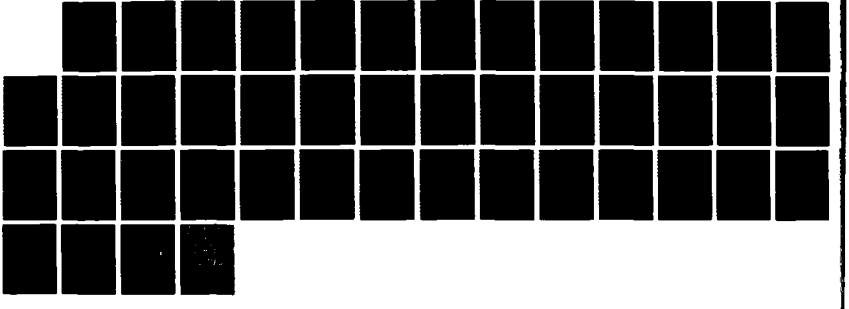
TABLE 5. COMPARISON OF THE LANGUAGES

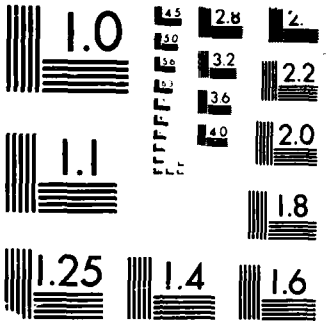
Language	AML	RCCL	AL	VAL	SRIL-90
ORIGIN	IBM	PURDUE	STANFORD	UNIMATION	IMPERIAL
COMPUTER	IBM Series1 mini	VAX11/780 Supermini	PDP 11 mini	LSI 11 micro	68000 micro
ROBOTS	RS1	PUMA/STANFORD	PUMA/STANFORD	PUMA	IMPERIAL
JOINT MOTION	Yes	Yes	Yes	Yes	Yes
ARM CONFIGURATION	-	Yes	Yes	Yes	Yes
CARTESIAN MOTION	Yes	Yes	Yes	Yes	Yes
TRANSFORMS	Yes	Yes	Yes	Yes	No
MOVING FRAMES	-	Yes	Yes	No	No
TOOL CHANGE	-	Yes	No	No	Yes
FUNCTIONAL MOTION	-	Yes	-	No	limited
VISION	Yes	Yes	Yes	Yes	Yes
FORCE FEEDBACK	None	Yes	Yes	No	user defined
COMPLIANCE	user defined compliance	Yes	Yes	No	user defined compliance
TOUCH SENSOR	(force/ switch)	-	-	-	-
OUTPUT CONTROL	Yes	Yes	Yes	Yes	Yes
COMPILED	-	Yes	Yes	-	-
INTERPRETED	Yes	-	-	Yes	Yes
DATA STRUCTURE	Any	Any	Any	Integer & transforms	Integer
STRUCTURED LANGUAGE	Yes	Yes	Yes	No	Yes
OPERATING SYSTEM	AML-IBM	UNIX	DEC	VAL	STROS
DEBUGGING	Yes	Yes	edit-time	Yes	Yes
IMMEDIATE OPERATIONS	Yes	Yes	Yes	Yes	Yes
WORLD MODEL	-	Yes	Yes	Yes	-

Source: Ahmad, Shaheen, "Robot Level Programming Languages and the SRIL-90 Language," Compsac 84 Conference Proceedings, Nov. 7-9, 1984.

AD-A193 796 HIGHER ORDER LANGUAGES FOR ROBOTS(U) MANUFACTURING TECHNOLOGY INNORMATION ANALYSIS CENTER CHICAGO IL J R BLAHA ET AL. OCT 86 MTIAC-SOAR-86-01 2/2

UNCLASSIFIED DLA900-84-C-1508 F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART

Bonner and Shin (5) present a detailed comparison of fourteen languages based on language features (Table 6). They further compared them by developing a sample program using the different languages. Their intent was to derive some measure of programmability. By performing the excercise, they actually identified entities at several layers which impacted the programmability (Table 7). These were: understandability, flexibility, and extensibility (language capabilities); compute power (operating environment); range of users or experience (programming environment); language structure; and variables and sensing (feature layer). Their analysis, then, at least hints that a comparison of languages based on language features alone is somewhat inadequate. Gruver, Soroka, Craig, and Turner (30) conduct a comparison which significantly begins to branch into other layers of the robot programming system. They compare eight languages (AL, AML, HELP, JARS, MCL, RAIL, RPL, and VAL) and break the comparison down into twelve categories:

Language Modalities (Programming Environment)
Language Type (Language Implementation)
Geometric Data Types (Language Features)
Display and Specification of Rotations (Language Features)
Ability to Control Multiple Arms (Language Features)
Control Structures (Language Features)
Control Modes (Language Features)
Motion Types (Language Features)
Signal Lines (Operating Environment)
Successful Sensor Interfaces (Language Features)
Support Modules (Programming Environment)
Debugging Features (Programming Environment)

Yong, Gleave, Green, and Bonney (6) provide a comparison of various programming environments (Table 8). Although they refer to this as a comparison of programming systems, it is, by the definitions used in this paper, a comparison of only a portion of the entire programming system.

Booze, Allen, and Hamilton (31) did a study to define the characteristics of a good robot programming system. One result of the study was a list of quality attributes affecting the life cycle cost of a robot program (Table 9). Since quality is a measure of the overall "goodness" of the language, it should not be surprising that the quality attributes are almost identical to what we have defined to be language capabilities. Another result of this

TABLE 6. COMPARISON OF ACTUAL FEATURES AVAILABLE IN THE 14 ROBOT LANGUAGES (page 1 of 3)

	Funky	T3	Anorad	Emily	RCL	RPL	Sigla	VAL	AL	Help	Maple	MCL	PAL	Autopass
Language basis	Programmed	Programmed	NC program	Assembler	Assembler	FORTRAN	Assembler	Assembler	ALGOL	Pascal and FORTRAN	PL/I	APT	Transform basis	PL/I
Language type	Compiler assembler, and interpreter	Interpreter	Interpreter	Assembler and interpreter	Interpreter	Compiler and interpreter	Interpreter	Interpreter	Compiler and interpreter	Compiler and interpreter	Interpreter	Compiler	Interpreter	Interpreter
Control structures	None	Condition execution on signals	J(seq) J<cond>	Branch, loopfor/ endloop, brcond	Goto BR** (** is condition)	Goto, if, do loops	If, ju	Goto, if/then, ifsig/ then	Begin/ end, while/ do, if/ then/else, etc.	if/then/ else/end, for/to/do, while/do, etc.	if/then/ else, do/end, goto, while/ do begin/ end, etc.	When... else endof/ when, while... endof/ while	For/to/ do, begin/ end	if/then/ else, while/do, begin/end, etc.
Callable routines	None	All sequences are routines	K(seq)	Include subr, return (p)	None	Call rtnsub (p)	All files are callable routines (p)	Gosub, return (no p)	Procedures, functions, macros (p)	Gosub, return	Procedures (p)	Extension, invocation, task, macro (p)	None	Procedures (p)
Nesting of routines	None	None	10 levels	4 levels	None	Yes	Yes	10 levels	Yes	None	Yes	9 levels	None	Yes
Data types available	Integer, hidden vector, scalar	Real	Real	Integer, array	Integer, point	Integer, real, common array	Real	Frame	Scalar, vector rot, trans frame, etc.	Real, ASCII array	Integer, real, array, frame, point, line, plane	String real, array, logic, frame	Matrix, array, scalar, stack	Integer, real, array, world model
Comments	None	None	G69	;	;	Comment ;	None	Rem	{....}	!	/*...*/	$$	None	/*...*/
Simple motion	Obtained by motion to points	Obtained by motion to points	x,y,z	Move, motor gopoint, joint motion	Draw move, appro, deprt, movea	Moveto	Mo	Go, move, movel, appro	Move with constraint clauses	Move	Move to/ by, rotate to rotate, by sweep to/by	From, goto, wkpnt	Mov	Move to move
Straight line motion	Only when using joystick	Only when using teach pendant	G1	Cartesian robot provides straight line	Draws, moves, appros, deprts	None	None	Moves, appros	None	None	Cartesian robot has straight line motion	Gofwd, goiff, gorgt	None	Push, slide

TABLE 6. COMPARISON OF THE ACTUAL FEATURES AVAILABLE IN 14 ROBOT LANGUAGES (page 2 of 3)

	Funky	IS	Anorad	Emily	HCL	HPL	Sigla	VAL	AL	Help	Maple	MCL	PAL	Autopass
Continuous path motion	Yes, Uses varying tolerances	Yes	G9	None	None	Smooth path control	None	Yes	Yes, "Via" gives intermediate points	Smove, FOM	None	Gofwd, goffl, gorgt	Yes	Implicit
Coordinate transformation commands	None	None	None	"Part" defines part frame	None	Tmult, tinver, twrite, tread	None	Shift, inverse, frame	#--> affix, unfix	None	Translated, rotated, fixed, relative	Usefrm, connect, discon	+(mult) -(invert and mult)	None
Gripper operation	Grasp, release	Uses external signal commands	M100-M195	Hand, dhand	Open, close, flip	Relayo, relayc	AX (external signals)	Open, Openi, Close, Closai	Open, close	Pulse, value	Open to, open by	Send/ hand open, send/ hand close	Gra, rel	Grasp, release
Parallel processing	None	None	None	Synch provides convergence point	None	None	AU (//exe), EW, ES wait, signal events	Signal, wait	Cobegin/ coend, signal, wait	Signal, wait, active, test	In parallel when event, begin... and	Inpar runs more than one task	None	In parallel, do
Interaction with external devices	None	Wait, output, external	M100-M195, M200-M223	None	None	Dtird, dtiwr, setox, cirox, rddxv, relayo, relayc	AX,PP (I/O signals)	Ignore, ifsig, react, signal, etc.	Signal, wait	Set, reset pulse, stroke	Status	Send, receiv, device	None	Switch, load, unload, fetch
Force/ torque feedback	A one-dimensional search	None	None	Tol, calib, getpos	Statsp	Inlarm	RP (exert), MT (read)	Calib	Force, torque	A<axis>, force	Torque, fleft-finger, fright-finger, fgap	Send and receiv	Compliance by equating axis alignments	Grasp, insert, push
Touch sensor commands	A command to center the hand on an object	Search command for stacking	M200-M223	Wand, sensor, getsen	Sensor	Dtird, dtiwr, dtini	TS and PP (test and set)	React, reacti, signal, wait	Center	If testb, test high	Hit, range	Send and receive	None	Insert extract, lower/ onto, lower

TABLE 6. COMPARISON OF THE ACTUAL FEATURES AVAILABLE IN 14 ROBOT LANGUAGES (page 3 of 3)

	Funky	T3	Anorad	Emily	RCL	RPL	Sigla	VAL	AL	Help	Maple	MCL	PAL	Autopass
Vision commands	None	None	None	None	None	Inivis, pictur, getfea, blink, delb, lo, recogn	None	None	None	None	None	Region, project, locate, inspec	None	Verify
Tool operation commands	"Tool" operates screwdriver	Tool status and external signal commands	None	None	None	None	External signal commands	"Tool" defines tool frame	Operate with constraint clauses	None	None	Tlaxis defines tool frame	Scr	Operate, fetch, replace, switch
Higher level commands	None	None	G2, G3 circular interpolation	None	None	Recogn identifies an object	IV, AV rel rel and abs motion; AC anti-collison	None	Compliant motion if force set to zero	None	Limits, setlimits	"Region" defines part dimensions	None	Place/on, drive/in, name/assembly, etc.

Source: Bonner, Susan and Shin, Kang G., "Comparative Study of Robot Languages," Computer, December 1982, pp. 84-85.

TABLE 7. QUANTITATIVE COMPARISON OF 14 LANGUAGES FOR A PALLETIZING-BLOCK PROGRAMMING EXAMPLE (page 1 of 2)

	Funky	T3	Anorad	Emily	RCL	RPL	Sigla	VAL	AL	Help	Maple	MCL	PAL	Autopass
Number of instructions	94	94	20	34(27)	76	58	35	35(22)	50	30	44(35)	69	25	8
Development time	1,5	1,5	4	2	2,5	5,6	4	2	3	2	3	6	7	1
Understandability of instructions	2	2,6	6,7	3,4	3,4,5	3,4,5	6,7	3,4,5	1,4	3,4	1,4	3,4,5	3,6	1,4
Structured format	3	3	3	1	2	3	3	3	1	2	1	2	3	2
Flexibility of variables	5	4	2	2	2	2	4	2	1	1	1	2	3	1
Ease of extension	4	4	4	1	4	1	4	4	2	3	2	3	4	2
Range of users	1,2	1,2	2	2,3	3	4	2	2,3	4	2,3,4	3	4	5	1
Programming complex tasks	4	4	4	4	4	2	4	4	3(1)	4	3	2	4	1
Computing power	1	1	1	2	1	2	1	1	2	2	3	3	2	3
Sensing ability	2	1	1	2	1	3	1	1	1	1	2	3	1	4
Availability	2	1	1	2	2	2	1	1	2	1	3	1	2	3

Source: Bonner, Susan and Shin, Kang G., "Comparative Study of Robot Languages," Computer, December 1982, pp. 84-85.

TABLE 7. QUANTITATIVE COMPARISON OF 14 LANGUAGES FOR A PALLETIZING-BLOCK PROGRAMMING EXAMPLE (KEY) (page 2 of 2)

Number of instructions. Of the different measures of programmability, the number of instructions in the program (excluding comments) is the only absolutely quantitative measure. Where two numbers appear, the first represents the actual number of statements, and the second (in parentheses) is the number of statements the program can be reduced to when extraneous assignments are removed. (Assignment statements are added to increase readability.)

Development time. This measure is divided into seven aspects:
(1) fast and simple
(2) quick, but requires some thought and/or intelligence
(3) quick, with background in structured programming and knowledge of transforms
(4) quick, but instructions are not easy to read
(5) encumbered by awkward control constructs
(6) encumbered by complicated coordinate transform usage
(7) requires extensive knowledge of coordinate transformation arithmetic.

Readability. Since readability is also quite varied and complex, it is divided into several submeasures that affect the readability of the languages: These include:

Understandability of instructions. Understandability can be characterized by seven aspects that relate to how the language reads.
(1) reads very much like English
(2) uses function button control
(3) instructions are words in English but do not read like English
(4) readability is markedly improved through proper choice of variable names
(5) instructions/variables limited to six characters
(6) instructions/variables limited to two or three characters
(7) instructions are not English-like at all.

Structured format. The degree of program structure in a language aids readability by enabling a known format for all programs and forcing variable declarations. There are three degrees of structure:
(1) structure is an inherent language feature
(2) can structure programs but not forced to do so
(3) the language is inherently unstructured.

Flexibility of choosing variables. Can the user define variable names? The following categories are considered:
(1) essentially unlimited variable naming
(2) variable names limited to six characters
(3) variable names limited to two or three characters
(4) variables available but only by number
(5) variables not available

Ease of extension. Four degrees of expansion are possible among 14 languages:
(1) subroutines or other extension facilities available
(2) expandable through subroutines with some loss of English-like syntax
(3) somewhat expandable through subroutines but nesting is prohibited
(4) must rewrite code to add instructions.

Range of users. Five levels of experience describe the range of users:
(1) novice
(2) NC programmer or machine operator
(3) person with some programming experience
(4) person familiar with structured programming and/or transformations
(5) person with extensive knowledge of transformations.

Programming complex tasks. The languages vary in their ability to program different tasks. Some are limited to very simple pick-and-place operations while others are capable of complex tasks such as fastening and compliant motion. This measure is not directly related to the programming example, but is included for comparative purposes. The task abilities of the languages are divided into four levels:
(1) capable of programming complex tasks using multiple arm operations, such as fastening
(2) capable of programming complex tasks using visual feedback, such as part recognition
(3) capable of programming complex tasks using touch and force feedback sensing, such as compliant motion
(4) capable of only simple tasks with minimal or no touch sensing in fingers.

Necessary support facilities. Measures of support facility requirements, while not directly related to the programming example are included in the table for a more realistic portrayal of which languages can actually be used today. The measures of support considered are:

Computing power. The language requires a
(1) microcomputer or small minicomputer
(2) minicomputer
(3) combination mainframe and small minicomputer.

Sensing ability. What robot capabilities are needed for the language's operation?
(1) only simple touch or no sensing required for complete operation
(2) proximity sensing specified in addition to touch sensing
(3) vision in the form of part recognition required
(4) complete dynamic world modeling system needed for operation.

Availability. Is the language as it is presented operational at present?
(1) operational and available commercially
(2) mostly operational but not commercially available
(3) only a small subset is operational.

TABLE 8. SUMMARY OF PROGRAMMING SYSTEM

FEATURES	AL (Stanford)	ANIMATE/PLACE (McDonnell Douglas)	AUTOPASS (IBM)	CATIA (Montpellier)	GRASP (Nottingham)	RAPT (Edinburgh)	ROBEX (Aachen)	GEOMAP (Tokyo)
GEOMETRIC MODELER								
Nonsolid	*	*				*		
Solid			*	*	*		*	*
Robot Modeler								
Kinematic		*		R	*		*	*
Path control		*		*	R		*	*
Generalized		*		*	*		*	
Programming System								
Manipulator level		*						
Object level	*		*	*	*	*	*	*
Textual			*			*	*	*
Graphical		*		*	*			
Sensor input	*		*		D		R	R
Other Features								
Reach testing		*		*	*		*	*
Control output	*		*		D	*	*	*
Dynamic collison			R	D	D		D	R
Cycle times		*		D	*			*
Multirobot	*				*			
Graphic simulation		*		*	*		*	*

* = Available; R = restricted; D= under development

Source: Yong, Y. F. ..., "Off-line Programming of Robots," Handbook of Industrial Robotics.

TABLE 9. IMPACT OF SOFTWARE QUALITY ATTRIBUTES ON LIFE CYCLE

	DEVELOPMENT		IMPLEMENTATION		OPERATION		
LIFE CYCLE PHASES / FACTORS	REQMTS	DESIGN	CODING	TESTING/ EVALUATION	OPERATION	MAINTENANCE	TRANSITION
CORRECTNESS	o	o	o	x	x	x	
RELIABILITY	o	o	o	x	x	x	
EFFICIENCY		o	o		x		
INTEGRITY	o	o	o		x		
USABILITY	o	o		x	x	x	
MAINTAINABILITY		o	o			x	x
TESTABILITY		o	o	x		x	x
FLEXIBILITY		o	o			x	x
PORTABILITY		o	o				x
REUSABILITY		o	o				x
INTEROPERABILITY		o			x		x

LEGEND: o Where quality factors are needed
x Where impact of poor quality is realized

Source: Review of the State-of-the-Art of Assembly Technologies and Programming Languages for Robotic Applications, Booze-Allen & Hamilton Inc., Arlington, VA., April 1982.

study was an identification of user requirements for robot languages (Table 10). A review of this list shows that these requirements reside in various layers of the robot programming system, which we have defined.

The study that comes closest to comparing the entire robot programming system was done by Collins, Palmer, and Rathmill (32) at Cranfield in England. They attempted to establish a benchmark for assembly. This involved developing an application, programming it, and then executing it on six different robots. From this test they concluded what types of entities are desirable to support assembly applications (Table 11). More importantly, the difficulty in the comparison became evident. A test of a robot programming system necessarily involves a test of the robot hardware. The language, programming environment, and operating environment, cannot be considered in isolation from one another other. The authors put it rather succintly:

> Being intentionally limited to Cartesian motions in X,Y,Z, and rotation around the Z-axis, the benchmark clearly tests the Olivetti Sigma robot more comprehensively than, say, the PUMA or IBM RS2. There is therefore an implicit problem of some importance in this direction, since benchmarks requiring more complex manipulator motions would leave a significant proportion of assembly robots unable to participate in the comparison....It would be valuable to see some work carried out, preferably using a suitable benchmark, which compares the performance of a robot programming language, such as LM or AML, when used in conjunction with a representative range of manipulators.

3.4 CAPABILITY COMPARISON OF SELECTED LANGUAGES

A comparison of languages based on language features can easily be done by referring to language manuals. From these, one can determine what features are supported in each language. A comparison of programming and operating environments is somewhat dependent on the actual implementation. For example, is a CAD/Graphics system available for simulation? What type of subroutine libraries is available? What type of robot arm is being used? The difficulties in this regard have been illustrated by Collins, et al. What is lacking in the literature is a comparison of languages based on their overall capabilities. We have attempted here to do such a comparison of some current languages. This subjective comparison is based on a study of the programming manuals and/or published material. Reference is made to specific language features as necessary to support a capability.

TABLE 10. USER REQUIREMENTS LIST FOR ROBOTICS LANGUAGES

• ON-LINE EDITING	• LOCAL AREA NETWORK INTERFACE
• ON-LINE TEACHING	• USER-DEFINED MACROS, SUBROUTINES
• INTERACTIVE CAPABILITIES	• MONITORING OF ONGOING PROCESSES
• GRAPHIC PRESENTATION OF DATA	• SIMULATION CAPABILITY
• TRACING OR SEARCHING	• FAULT TRACES AND EASE OF DEBUGGING
• MULTIPLE ROBOT PROGRAMMING	• DATA CODE STANDARDS
• MODELS OF PARTS GEOMETRY	• CONCISE PROBLEM FORMULATION
• HIERARCHICAL/STRUCTURED CODE	• CONVERSATIONAL/ INTERACTIVE
• ENGINEERING FAULT ISOLATION	• ADAPTABLE TO INDUSTRY JARGON
• FAILURE RECOVER	• SENSORY INTERACTION SELECTABLE
• OPERATOR INTERVENTION	• SEPARATION BETWEEN SENSORY ANALYSIS AND CONTROL
• MACROS WITH ROUTINE PARAMETERS	• SIMPLE GENERATIVE PLANNING
• ADAPTABLE TO STRUCTURED TECHNIQUES	• FAST SERVO ALGORITHMS FOR END-EFFECTORS WITH COMPLEX DYNAMICS
• INTERFACE WITH DBMS/CADAM AND CLDATA FILES	• SUPPORT OF MATERIAL FLOWS TO/FROM THE ROBOT WORK AREA
• DATA AND INPUT TEXT PARSING AND EDITING	• ACCURACY IN WORD SIZE TO .001 INCH
• COLLISION AVOIDANCE	• MULTITHREADING FOR CRITICAL PROCESSING TASKS
• ASYNCHRONOUS ROUTINES DRIVEN BY EXTERNAL OR INTERNAL EVENTS	• EASE OF UTILIZING NEW APPLICATIONS
• IMPROVED VISION/SENSOR DATA	• PORTABILITY
• MODEL SPATIAL RELATIONSHIPS	• INCREASED FLEXIBILITY IN DEFINING CYCLE LOGIC

Source: McDonnell Douglas

TABLE 11. ASSEMBLY RELATED ROBOT LANGUAGES AND PROGRAMMING FEATURES

ESSENTIAL	DESIRABLE	ADDITIONAL
1. Straight line motion in at least one direction	5. Straight line motion in any direction	13. Acceleration control
2. Gripper control	6. Servo control of gripper	14. Vision facility
3. Response to external signals	7. Tactile sensing	15. Software maintenance checks
4. Originate output signals	8. Step speed control	16. Control of 'settling' time
	9. Editing facility	17. CAM compatible
	10. Computational ability	18. Produces a listing if printer is interfaced
	11. Program back-up facility	19. Diagnostics
	12. Program decision making	
	20. Direct Position teach facility	

Source: Collins, K. ...., "Development of a European Benchmark for the Comparison of Assembly Robot Programming Languages: A Comparative Evaluation"

3.4.1 VAL and VAL II

VAL is a manipulator level language, having a BASIC-like structure, designed for Unimation robots. The PUMA manipulator is connected to the VAL monitor, an LSI-11, which runs its own operating system containing an editor, file manager, and user interface. The design intent of VAL was to allow an operator to define "tasks" as a series of moves and gripper actuations.

VAL was originally designed by Bruce Schimano as the VicArm Language, as it was to be used to control the VicArm created by Victor Scheinman. When VicArm became the West Coast Division of Unimation in 1977, a contract was initiated with GM resulting in a prototype for the PUMA robot in 1978, to be principally developed by Scheinman and Brian Carlisle. The language was changed to Versatile Assembly Language by Unimation. Today, VAL continues to lead the industrial market in the number of installations.

A VAL program consists of two parts. The motion sequencing portion can be developed off-line, as a collection of subroutines corresponding to movements. This part is devoid of actual coordinate and speed details. The detailed Cartesian coordinates specifying where the endpoints of a move are located and the speeds involved in the trajectory are provided through an on-line teach session, which fulfills the second portion. This is done by stepping through the program in a combined teach/manual mode, jogging the manipulator to the physical points referenced by the program and then copying in the actual Cartesian coordinates and speeds to the program memory.

FLEXIBILITY. The VAL language is heavily motion-oriented, yet provides some of the power of BASIC programming. Subroutines are supported, without parameters, and the coordinate system can be designated right- or left-handed. It supports frames, joint angles and transformations, but lacks iteration, conditional, and block structures.

EXTENSIBILITY. A major revision, VAL II, has been released and offers considerably more flexibility in motion control, with bias forces, guarded moves, real-time trajectory updating, watchdog monitors over sensors, and tool frames.

RELIABILITY and CORRECTNESS. The use of the teach mode togetner with a textual program helps take the guesswork out of the first few runtime debugging sessions. However, exception handling mechanisms are not a feature

of VAL, so runtime failures may be a problem. The lack of sophisticated conditional branching makes explicit error handling difficult.

MAINTAINABILITY. Since the motion sequencing is kept separate from the location and trajectory data, VAL programs are somewhat reusable. By teaching a different set of points, the same generic motion sequence program may be used to handle differing parts.

EFFICIENCY. The original VAL was an interpreted implementation using C and assembly code on an LSI-11. VAL II, however, is much faster and allows for multiprocessing.

3.4.2 RAIL

Designed by Automatix Incorporated, RAIL was intended to supply a flexible interface between manipulators and vision systems. The approach taken was to create an integrated environment with the strongest hardware available. The result was the AI 32 controller using a Motorola 68000, running an operating system interfacing RAIL programs, robot control software, and image processing software.

The programming environment can be off- or on-line, with a fair amount of tools provided within the operating system. An editor, file system, serial/parallel communication, and multiprocessing are supported. A RAIL program is a procedural program which is translated into an intermediate code before actual execution by the interpreter. The translation is transparent to the programmer; it is not a separate development stage. RAIL source programs have the structure of PASCAL, with a good complement of control and conditional constructs. A sophisticated pendant, called the Interactive Command Module (ICM), can jog the robot in joint, world or tool coordinates. It also can define locations and frames, paths, speeds and serves as a simple editor. (34)

FLEXIBILITY. RAIL is quite flexible, having a number of general purpose PASCAL-like constructs in addition to motion, image processing, math, and I/O commands. There is good support of regular and geometric data types such as frames, paths and transformations.

EXTENSIBILITY. The integrated nature of the operating system should enable RAIL to extend as new sensor technology becomes available.

RELIABILITY and CORRECTNESS. The translation stage catches syntactic errors before execution and the early stages in program development are enhanced by the "teach-by-showing" vision method, analogous to teach pendant programming.

MAINTAINABILITY. The PASCAL-like syntax is rather self-documenting. Modular development also enhances its maintainability.

USABILITY. The integrated approach taken in design is responsible for the success of the RAIL language in applications requiring heavy sensor support. The system has found acceptance in welding, inspection, and assembly.

3.4.3 KAREL

KAREL is a manipulator level language consisting of a translator and an interpreter, written by GMF Robotics for their line of controllers. A KAREL source program resembles a PASCAL or MODULA program in many ways. However, a number of built-in functions and procedures for math, I/O, character manipulation, robot motion, and line tracking give pre-defined methods for real world control, which would have to be extensions to more general purpose languages. The design approach was to take a strongly structured information processing language and add the necessary features to perform motion and process control, sensor integration and communications.

A KAREL program can be written on the controller or off-line and downloaded. A syntax directed editor is provided that "knows" the permissible KAREL constructs, so program creation and maintenance is enhanced. The resulting source file is translated to a lower form, which is then executable by an interpreter on the controller. The translator uses static scoping rules, which promotes reliability and maintainability, while the interpreter gives flexibility at execution time. Modular program development is enabled because of the translation stage and because a program module may refer to external procedures or entire files. This exemplifies the strength of the design approach. (21)

FLEXIBILITY. KAREL is a very flexible language, because it includes all the elements normally associated with a computer language and specific language additions pertinent to robotics and manufacturing. Procedures may be passed parameters by reference or by value, and they may be locally defined or accessible in external files. The same holds true for functions. Because of static scoping, which mandates liberal use of declarations, recursion is also possible. Vector operators and relative motion are also supported, as well as access to the command level (operating system) of the controller.

EXTENSIBILITY. The combination of translation and interpretation used to create executable KAREL programs makes it easier to extend the language by simply revising the translator. Postprocessors could take a very application specific front end and convert it to a normal source program for the translation stage. It may be possible to write a custom translator in order to combine these steps.

UPWARD COMPATIBILITY. Revisions to KAREL will probably entail language extensions to accommodate users and should only require the issuance of a revised translator. Old KAREL programs should therefore run acceptably on newer versions.

PORTABILITY. The low-level KAREL interpreter is currently available only for their controllers, but the translator should be available for general purpose computer systems.

RELIABILITY and CORRECTNESS. The translation stage assures than any runtime errors will not be syntactic. KAREL is also notable for its exception handling capabilities, which may be locally defined or global to the controller. These handlers are actually condition monitors which are checked every 32 milliseconds. By prioritizing them, a certain extra bit of reliability is gained under failure conditions.

MAINTAINABILITY. The data abstraction and modularity gained through the design approach provide KAREL with good maintainability.

EFFICIENCY. Because of the significant built-in language extensions applicable to real world control, the executable code should be efficient. Apparently, the interpreter runs at comparable speeds to other controller makes.

range of robots and sensors. AML/2 defines (and implements) such motion and sensor primitives, as well as application control and operator interfaces, for the IBM 7575 and 7576 robots.

EXTENSIBILITY. The freedom for data abstraction allows the user the continual ability to extend the capabilities of a system, by placing more user-oriented "front ends" on top of an application, or by integrating new sensors, robots, and information sources into the application.

UPWARD COMPATIBILITY. AML is somewhat compatible with AML/X. Some programs written in AML will not work with AML/X because of design decisions related to the future implementations.

PORTABILITY. AML/X is very portable, since the interpreter is written in the C general purpose language. It has been ported to IBM 370, IBM PC, and Motorola 68000 running under CMS, UNIX, XENIX, and DOS. Low level servo control software routines callable from AML/X are under development. Such routines are included with AML/2 for the IBM 7575 and 7576.

RELIABILITY and CORRECTNESS. As with any language, correctness depends on the program and how well it reacts to changing conditions. In some ways, an AML/X program may be more reliable than an off-line translated program, since the exception handling mechanisms are extremely sophisticated.

MAINTAINABILITY. The data abstraction and rather self-documenting style of AML/X contribute to its maintainability.

EFFICIENCY. As the current release is an interpreter, it will not be feasible to write some applications solely in AML/X. However, time critical parts of an application can be written in C and called through AML/X's C interface.

DECISION MAKING. The language has all the power expected of a modern, general purpose programming language.

USABILITY. Program development under an interpreter is always easier than with a compiler. The fact that AML/X is devoid of robotic terms like "move" and "grasp" means that an application shell defining such terms needs to be designed by an application developer before "robot programming" can be done by an end user. AML/X, with its object-oriented features, is well suited for building such an application shell. Indeed, AML/2 provides a shell

containing facilities for the robot programmer. It includes commands for various kinds of moves and guarded moves, monitors, and grasping. An AML/2 application simulator and an editor that knows AML/2 syntax are also available to further simplify robot application programming.

3.4.5 Results of Comparison

The four languages VAL, RAIL, KAREL and AML/X have been reviewed according to the language capabilities developed in Section 3.2.3. As far as was possible, each capability was evaluated according to its component language features, programming and operating environment, and syntax. The set of capabilities chosen was discovered to be nonorthogonal, which is to say that they were dependent on each other in some cases. A valid set of criteria for an evaluation should, insofar as possible, be mutually independent. This helps to ensure that all issues are examined with equal rigor. For example, the extensibility of a language depends to a large degree on its flexibility; likewise, upward compatibility is related to extensibility. These criteria measures are therefore correlated, and perhaps should have been combined into more generic measures or broken down into independent components.

This exercise was done to illustrate our viewpoint that the programmability of a robotic system depends on all of the factors involved in the process of programming, not just the language features. Our particular choice of language capabilities, as criteria for evaluation, was based on the sum total impression gained through our research and interviews. Consequently, the reader may find them skewed from other familiar comparisons. Nevertheless, the macroscopic viewpoint is a sound one, particularly in the beginning stages of search for a suitable language.

3.5 ROBOT PROGRAMMING ISSUES

A number of issues in robot programming languages have been discussed, and these are reviewed and expanded upon here. When comparing various robot programming languages these issues should be kept in mind. The issues concern entities in the layers of the programming system. The approach taken by a particular language on a particular issue will therefore influence the other layers and, ultimately, the overall performance of the programming system.

3.5.1 Teach Programming vs Off-line Programming

This is an issue in the programming environment, arising from a disagreement as to the type and level of experience required of robot programmers. The relative advantages and disadvantages to each programming technique are discussed at some length in Section 1.6. One side of the argument is that applications people on the shop floor should program robots. These people are familiar with the environment and are most aware of the problems involved in performing a task. In addition, a great many tasks can be programmed more efficiently with teach pendant programming than with off-line methods. The other side of the argument is that complex tasks, such as those generated in a flexible manufacturing environment, require off-line programming techniques. In such cases, the ability to integrate computers into the design, manufacture, and test of products would be limited by teach pendant programming.

This argument should not center so much around teach pendant and off-line programming techniques as it should around programmer experience. Off-line and teach pendant programming techniques are complementary, not competitive--each has applications for which it is the most efficient method. The real issue is one of user-friendliness. The question is whether trained computer programmers should program robots directly, or whether they should provide application programs for use by application personnel. Application personnel can write off-line programs if provided with tools that allow them to communicate in the language of the application. Interactive CAD/Graphics packages take this approach. User companies will probably each take their own approaches, some electing to have computer programmers become robot programmers and others electing to have computer programmers develop a set of user-friendly tools for application programmers. Vendors will be in the position of trying to satisfy both. In order to do this, they will have to provide user-friendly languages that also allow access to low-level constructs. As languages evolve toward task level, this will probably become less of an issue. Programmers at all levels of experience will be able to generate robot programs. There will always be cases, however, where users need access to low-level constructs so that a programmer can explicitly direct some robot actions.

3.5.2 Language Approach

Three approaches to this issue have been identified. The first is an extension to NC languages, such as that taken by RAPT. The second is an extension to existing computer languages, flavored for robotics, such as that taken by KAREL. The last is a totally new language development, such as that taken by AML. The issue is complicated a bit by using AML as an example, because some may agree that a totally new language approach should be developed, but may not agree with the particular approach taken by AML.

There are really two issues here. Once again, user-friendliness is a factor. One argument for the use of NC-based languages is that the functional languages are more user-friendly and that there are many experienced APT programmers. The same argument would oppose AML on the grounds that it is just too difficult to learn. It might even go so far as to say that a new language is needed that is more user-friendly than the NC-based languages.

The second, and more important, issue is that the entire robot programming system is impacted by the approach to the language design. The relative strengths and weaknesses of the different approaches as they relate to requirements of the robot programming system must be considered.

The advantage of NC-based languages is that they are good for motion control and for manipulating objects. The disadvantage is that they are not good for sensor integration and information processing. As languages evolve toward the task level, applications will become more complex. This will involve significant sensor integration and information processing. The robot controller of the past was basically a motion controller which did very little of anything else, similar to an NC controller. This is why APT-based languages seemed a reasonable approach. The robot controller of the future, however, will be on the opposite side of the spectrum--it will basically be an information processor which does comparatively little motion control. For this reason, many have taken the approach of extending computer languages.

Whereas the NC-based languages have been motion- and geometry-oriented and have been extended to included information processing tasks, the computer-based languages have been information processing oriented and have been extended to include motion and geometry. In Section 3.2.7 robot language

features are discussed. These have been distilled to isolate the extensions required for robot control, and are summarized as follows:

Geometric Data Types
Special Arrays (Frames, Matrices)
Matrix Operators (Affixment, Transforms)
Coordinate Systems
Vision Commands
Compliance
Timers
Motion Control Commands
Tool Statements

The approach of extending a conventional computer language appears a viable one. The extensions are mostly expansions to existing feature classes in conventional languages. The base language to be used is yet another issue. A number of current manipulator level languages are based on PASCAL or BASIC. Robot languages have typically evolved in a pattern similar to conventional computer languages. The next generation of languages is likely to be based on a language which supports data abstraction and symbolic communication. Included here are Ada and the artificial intelligence languages such as LISP or PROLOG.

The approach to creating a totally new language is also viable. By doing this, language developers can concentrate on the requirements for the entire programming system. Rather than extending feature classes of conventional computer languages, the entire feature class can be designed from the ground up with robotic applications in mind. This can result in the significant power and flexibility required for robot programming.

3.5.3 Extent of Task Level

As languages approach the task level, tradeoffs are made. The ease in programming at a task level is achieved at the expense of providing significant computer power in the robot controller. This was illustrated in the discussion of decision making. Some amount of task decomposition by the programmer can greatly alleviate computational burden on the controller. The computational abilities of the controller have practical limitations of speed and cost. A truly task level controller will not be practical for many years. An issue to be resolved is just how high a task level is really required.

Although the trend toward task level seems certain, a great deal of benefit may come from development of strong application packages analogous to application packages in the conventional programming domain. Section 1.3 showed the evolutionary stages in computer language development, which have a counterpart in the CAD/Graphics packages of Section 1.6.4. Application languages are a step in the direction of task level languages.

Since computer power is typically more abundant in the programming environment than in the operating environment, task level languages are likely to be pioneered on off-line workstations as application programs. Task level commands will be decomposed by a postprocessor into robot controller commands. As these commands become well established, software will be developed so that they can be input directly into the robot controller.

3.5.4 Robot Control vs Cell Control

When robots are incorporated into manufacturing cells, the robot controller is often the most powerful computer and the robot language is the most flexible programming method in the cell. Since the robot program must be synchronized with other equipment, it is natural to use the robot controller as the cell controller, and to slave equipment off the robot program. This, of course, depends on the particular application. Some will have one robot and two small pieces of equipment; others will have many robots, machine tools, and sensors. The latter case is likely to have a cell host computer with a master program, off which all other equipment is slaved.

The issue here is the proper scope of robot programming languages. Should these be expanded to cell control? If their scope is expanded upward to cells, should it also be expanded downward to machine tool control? There are no clear answers here, since the application is really the driving force. What can be said is that perhaps "scope" should be added as an entity in the application layer of the robot programming system, and should routinely be considered when evaluating languages.

3.5.5 Standards

There has been some talk of standardizing robot languages at various programming levels. Some vendors would like to have their own languages established as industry standards at the manipulator level. Meanwhile, a servo level standard has been proposed in Germany. (4) This is the Industrial Robot Data (IRDATA) proposal. One could also envision a task level language someday acting as a standard. In fact, a standard format could be envisioned at any level (all five levels proposed by Bonner and Shin, for example). The general consensus of most of the experts interviewed is that the technology is not mature enough for standardization. Development in sensor technologies and a predicted change away from the use of the traditional Von Neumann computer architectures are likely to have a large impact on robot programming languages. Also, experience must be gained over more applications, so that these requirements can be incorporated into any standard.

The real push for standards may actually come when languages approach the task level. As things become specified at higher levels there is a greater chance for ambiguities and thus a need for standardization. For example, "put box A on box B" means something different from "put nut A on bolt B." Is the meaning to be derived from the word "put" or from the objects specified (nut and bolt vs box and box) or from a combination of the two? Is specification of the task application dependent? For example, does "put" mean one thing for assembly and another thing for arc welding? At the task level, some standardization will be desirable to establish the exact intent of each specified task.

3.6 SUMMARY

Comparison of robot programming languages is difficult because the subject area is very complex. The literature fails to make clear several important considerations and distinctions. First, the robot programming language must be put in perspective as a factor in overall robot performance. Second, the difference between robot programming techniques and robot programming levels must be established. Third, the difference in robot programming requirements at the task level and at the manipulator level must be noted. Finally, the entire robot programming system must be considered.

The robot programming system was shown to consist of four "layers." The outermost layer is the language capabilities layer, next the programming/operating environments layer, then the language implementation layer, and finally the language feature layer. Each layer consists of a set of entities, which are factors relevant to that layer. In general, entities in the outer layers give rise to requirements for supporting entities in the layers below. Two layers outside the programming system, the application layer and the industrial environment layer, are seen to impact the system. A certain application requires specific language features to support it, but also establishes needs in the other layers. A proper comparison of robot languages requires that the entire robot programming system be considered.

There are several robot programming issues that have not been resolved. Most revolve around the question of how user-friendly robot programming should be, and who should program robots. These are the issues of robot programming techniques, approach to robot programming language design, and the extent of task level programming needed. The approach to language design is, also, and more importantly, an issue of supporting entities in the robot programming system. Other issues are the scoping of robot languages (robot vs cell) and standards.

4. ROBOT PROGRAMMING LANGUAGES--A DoD PERSPECTIVE

Evaluation of robot programming systems for defense manufacturing can be performed using the criteria set forth in Section 3. Defense manufacturing is characterized by a few considerations. First, lot size is usually small and it may be necessary to supply the parts in several shipments over a period of time. Second, many subcontractors usually work on a project, and there needs to be communication among them and the prime. Third, military standards must be upheld and there is some amount of accountability (record keeping) that must be maintained. Finally, on a different note, much of the manufacturing is actually rework of old or damaged parts.

The implications of batch manufacturing have been discussed. Batch manufacturing requires frequent reprogramming. The language capabilities required to support this are flexibility, efficiency, and sensor support. The robot programming system requires a strong programming environment with a good editor, off-line debugging, and simulation. Also, maintainability is very important if a program is to be re-used at discrete time intervals.

With many subcontractors working on a job, it is possible that programs will need to be shipped between them. This would require a language with portability and good documentation. This may also argue for a standard language.

Good record-keeping also requires good program documentation. In addition, reliability, correctness, and error logs are important. This raises an interesting point. All robot languages are communicated in only one direction; that is, the communication of robot programs is always from the development system down to the robot. Military applications may require that languages also support communication the other way. A report on the current status of the workcell may be important. A process trace that records the exact events occurring in the workcell during manufacture of a part may also be needed. The robot language could support such record-keeping messages in its own syntax. This bi-directional language concept has been used by Computer-Aided Manufacturing-International (CAM-I) in developing the Dimensional Measuring Interface Specification (DMIS), a programming language for inspection devices.

The rework of parts presents a particularly interesting application. CAD models of parts are rarely available, and the exact program that needs to be executed by the robot depends on the particular repair to be done. This type of application requires significant sensor support and a very strong programming environment. Expert systems and other AI technology will be required before automated operation is possible. In the meantime, a combination of teach pendant programming and off-line programming will be required. Teach pendants can be used initially to generate the part geometry and some information about the defect (the location of a crack, for example). Off-line techniques can then be used to generate a sensor driven robot program capable of performing the gross motions necessary to approach a feature. The robot would then be guided by sensors to perform the detailed repair operation.

5. RECENT AND ONGOING RESEARCH AND DEVELOPMENT

In the early 1980s, a number of good papers on robot programming languages were published, but very little has been written since 1983. In these papers, manipulator level languages were compared and the technical barriers to task level programming were explored. These technical barriers still exist, and some research efforts are being directed toward solving them. Other efforts are being directed at improving the state-of-the-art of robot programming with available technology. For purposes of discussion, it is convenient to break this down into four topic areas: servo level, manipulator level, task level, and applications of artificial intelligence.

5.1 SERVO LEVEL

While servo level languages have, in general, not been the subject of much research, there are some significant efforts which should be cited. As mentioned in Section 1.5.1 some vendors are committed to increasing the power of teach pendant programming, and are carrying on with in-house developments in this area.

The IRDATA proposal in Germany is an attempt to standardize robot language at a low level. Since it is being proposed as a standard, it will undergo substantial industry review. Review processes such as this tend to raise and investigate issues otherwise overlooked.

Greshke (35) proposed the Robot Servo System (RSS) as a servo level language which could act as a target language for manipulator level programs. The approach taken in this language was to specify everything in terms of servo processes. Data from external sensors was represented as dynamically changing functions. The advantage of the language was that the programmer could control what sensor data and robot state information was to be used by each and every servo process.

Ford Motor Company is currently developing a system which combines a teach pendant programming technique with an expert system. Using a specially designed teach pendant, the programmer (operator) teaches the robot only a few points. He does this by following a set of instructions and selecting options from a menu. The system then computes all of the intermediate points necessary to drive the robot through the complex path, and outputs a servo level program specifying the motions.

Investigating an interesting concept, the University of Connecticut (36) developed an experimental system called Task Learner (TL). Here a servo level program is used to generate a task level program. The programmer guides the robot through a task using a teach pendant. The resulting servo level program is then used to infer the task plan, which can later be repeated in a variety of workcell configurations.

5.2 MANIPULATOR LEVEL

Work on manipulator level languages and the next generation of robot languages is continuing mostly in the vendor community. The main thrust is to increase the power and flexibility of current languages, and to provide object-oriented programming. The advancements evident in VAL-II over VAL-I provide an indication of the trends in robot programming technology at the manipulator level. Shimano (37) has summarized these as follows:

- formal network communication facilities
- mathematical capabilities equivalent to those in high-level computer languages
- enhanced operator interface
- extended sensor capabilities
- real time path modification based on both internally and externally generated command signals
- facilities for performing simultaneous control of both robot and process activities.

Languages such as VAL-II, ARLA, KAREL, and AML/X reflect the state of the art in vendor-supplied robot programming languages. Steady progress will be made toward realization of task level languages, and some capabilities are emerging. The AIMPCB language developed by Adept, Inc. is often cited as an example.

MCL was developed by the United States Air Force as a language for robot and cell control. In a continuing R&D effort to test the language, three aerospace firms are currently using MCL: Fairchild Republic Company, Grumman Aerospace, and McDonnell Douglas. Several problems have been uncovered. First, an MCL program is hard to simulate on a CAD system. Second, collision detection is very difficult because of the MCL technique of user frames. Finally, development time for an MCL program is two to three times longer than that for other languages. There have been recommendations to continue testing and to develop a generalized simulation system for MCL. These types of R&D efforts are beneficial, since the results can be transferred to private industry. For instance, many of the ideas that came out of MCL development have been incorporated into commercial languages.

Volz and Mudge (11) (12) (13) have proposed Ada as an advanced manipulator level, bordering on task level, language for use on robots and in manufacturing cells. They list the advantages of Ada as data abstraction, incremental compilation, multitasking, exception handling, timing constructs, modularity, extensibility, maintainability, and usability. It can be used as a base language for building applications programs and can access CAD data bases. There are some corresponding disadvantages in the additional computational overhead associated with extremely powerful languages. Initial work has looked promising and research is continuing.

5.3 APPLICATION LEVEL

An evolutionary step in the progress toward task level languages is shown by application programs. Recall that computer application programming such as spreadsheets, accounting and planning programs are essentially very high-level language processors, which accept the statement of a problem in the language suitable to the domain of discourse. The user specifies the problem in terms of costs, rates and time, which are then interpreted and solved by a program developed in some lower level conventional computer language. Robotic programming seems amenable to a similar approach. In this case, the user would state the problem in terms of goals, constraints, objects, features, and processes to an application package which would generate a program capable of running on a controller. Current CAD/Graphics systems address the surface of

this programming technique by providing good user interfaces for the specification and analysis of trajectories. What still remains to be developed is a comprehensive approach to the design of such an applications package. Work under way at NBS is promising in several of these areas.

5.4 TASK LEVEL

Task level languages are many years away due to several technical barriers. All of these are the subject of ongoing research and development to some degree. The key thing to remember is that task level languages are sensor driven. This gives rise to the need for world models of what should be (programmed goals), world models of what is (sensor input), and strategies to make the two equivalent using path planning, grasping strategy, and decision making. (38) (39).

World modeling is the problem of accurately representing the environment within which the robot is operating. Solid modeling techniques will be required to represent surfaces, volumes, and many part features. The concept of uncertainty must somehow be represented in the world model. Uncertainty arises from imperfect accuracy and repeatability, temperature changes, the dynamics of motion, variability in parts, robot drift, unexpected events, and changes in the world state between sensor readings. Sensors can compensate to some extent, but sensors also have some inherent inaccuracies and therefore introduce their own uncertainties. Two methods for representing error as it propagates through tasks are numeric error propagation and symbolic error propagation. Another problem with world modeling is in relating nongeometric data to objects. For example, the final torque on a bolt should be part of the world model. There is also a problem of overspecifying the world state in some cases. For example, if a pin is placed in a hole, the rotation of the pin is irrelevant and should not have to be specified. If it is not specified, this is known as a point of singularity, and the robot is faced with an infinite number of solutions. Strategies for selecting one must be developed.

Path planning represents a very complex problem. Modeling objects in three-dimensional space, so that a collision-free path can be defined, is extremely difficult. Different solutions to this problem have been investigated and are discussed in the literature (40). If objects are moving, the problem is complicated in that computational speed becomes a constraint and

three-dimensional vision sensing must be used to update the model. An additional problem is that the path is affected by initial orientation, final orientation, and speed. Therefore, path planning cannot be done until a grasping strategy has been selected.

Grasping strategy involves computing the appropriate approach and orientation for picking up a part. This requires not only that the geometry of the part to be picked up be considered, but also that any potential interference with surrounding objects be taken into account as well. Nongeometric information associated with the part, such as center of gravity, orientation constraints and coefficient of friction, must also be considered. One problem in grasping strategies is to select the appropriate option from a list of possibilities--there are often many ways to pick up a part. Some criteria, such as stability, must be set and then evaluated. Another problem is that of grasping under uncertainty.

Decision making currently involves branching to some part in the program when an error condition occurs. This often involves attempting the task again ("if part not present, then try to pick up again"). This does not deal with two error conditions which must be solved for task level operating environments. The first is unexpected errors. The robot must be able to recover gracefully from an error which was not anticipated at programming time. The second condition is an irreversible error. In this case, the robot must be able to go ahead and recover from the error by performing some other series of tasks.

Current work at the National Bureau of Standards' Automated Manufacturing Research Facility addresses a number of these issues. A robot programming system called RCS is being developed using the FORTH computer programming language. RCS breaks down the robot programming task into six levels: task, subtask, primitives, e-moves, gripper commands, and communications. Note this is not unlike the robot programming levels discussed in Section 1.5. The issues being addressed are geometric reasoning, component reorientation, automatic grasping, collision detection, path planning, automatic fixture design, mechanical testing of fixtures, and graphic simulation of fixture layout. Geometric reasoning is the description of an assembly task by the relationships between features on objects. An example would be to "place face A against face B." The principles of group technology are applied and objects

are described as instances of classes. For path planning and collision detection, the robot environment is modeled using a spatial tree representation. Several planning algorithms are then used to generate a path within the constraints of this model.

5.5 APPLICATIONS OF ARTIFICIAL INTELLIGENCE

Artificial intelligence technology will significantly impact the area of robotics programming. Task decomposition, planning strategies (path and grasping), and error recovery are all natural applications for artificial intelligence. There are some existing applications and research efforts are currently underway.

In a commercial application, expert systems are being used for diagnostics on ASEA robots. The expert system generates a set of instructions for tracking down failures in the robot controller. Currently, this amounts to incorporating a troubleshooting manual into the expert system. A more intelligent system with a larger ruleset may be developed in the future.

Westinghouse (41) has developed a workcell to forge steam turbine blades for electric power generators which uses a rule-based control system (expert system). It consists of two vision-guided robots, several NC machines, and a rotary hearth furnace. Setup and operation are completely automated, and the cell automatically adjusts for different blade batches.

Under a current contract from the United States Air Force, Martin Marietta is developing the Intelligent Task Automation (ITA) cell. In the project, all of the computers, except for the servo drives, were stripped from a Cincinnati Milacron T3 robot controller. These were replaced by single board computers communicating with a Symbolics AI machine using Ethernet. The system is used for performing inspection with a robot. The user simply indicates what points are to be checked, and the required robot program is automatically generated. The system consists of two parts: an AI planner and a path planner. The AI planner is a rule-based system which does tool selection and generates a sequence. The path planner does incremental and global path planning for collision avoidance under various constraints. It also monitors execution by comparing a world model with information acquired from sensor input. (42)

6. RECOMMENDATIONS AND CONCLUSIONS

6.1 RECOMMENDATIONS FOR RESEARCH AND DEVELOPMENT

Based on analysis of the literature and interviews with industry professionals, the following have been identified as areas requiring research and development:

1. Methodology for Robot Programming Languages. Efforts should be directed at an understanding of the "big picture." We need to understand the requirements of various applications and the information flows necessary to automate them. A data model of robot programming would be beneficial. Also, we need to identify which parts of the process are feasible to automate and which parts are better left to human reasoning.

2. World Modeling. Better techniques are required for representing and updating computer models of the robot, the environment, and the associated processes.

3. Simulation. Better simulation techniques, including automatic interference checking, kinematic and dynamic models, and sensor simulation, need to be developed.

4. Communication. Better methods of communicating information among equipment in a complex process must be developed. This includes both the content of the information and the speed with which it can be passed.

5. Sensor Technology. From a software viewpoint, methods for acquiring information from a large array of sensors and then processing it into a consistent world model are needed.

6. Safety. Safety considerations, particularly for task level programs, need to be addressed.

6.2 CONCLUSIONS

Much work has been done in the development of higher order languages for robots. There seems, however, to be no model which correlates all of the elements of the robot programming system. Languages do not operate in isolation: programs are developed in a programming environment and are executed in an operating environment. Languages are required to support both of these

environments, as well as the other layers of the system. A top-down systems approach to developing the requirements of robot programming systems is required.

The evolution of robot programming systems has been shown to parallel that of computer programming, with a lag time of five to ten years. This observation has also been made by previously cited investigators at the National Bureau of Standards, who further suggest (in a forthcoming paper) that this knowledge be put to use. A number of recent advancements in computer programming can, and perhaps should, be put to use in robotic programming. The present robotic programming inadequacies are largely problems of information processing, program development, and program testing. Existing systems have solved earlier motion control inadequacies with faster, and more parallel, hardware. The state-of-the-art computing technology should be applied directly to these three problem areas.

Much of the debate in the industry revolves around the user-friendliness of the programming language. There are two questions to consider about user-friendliness: what is it, and where is it ? First, user-friendly is a subjective term. What is user-friendly to a computer programmer is certainly different from what is user-friendly to someone with no computer language experience. Furthermore, what is user-friendly today is not the same as what will be user-friendly in five to ten years. Since computers are now introduced at elementary school levels, we can expect the whole concept of "user-friendly" to be quite different in the near future. Second, user- friendly front ends are finding their way into CAD/Graphics systems. As these front ends become commonplace, there will be less impetus to make the actual manipulator level languages themselves user-friendly. Robot vendors may even develop user-friendly front ends for their own manipulator level languages. That is, the vendors may develop one language for the programming environment and a corresponding language for the operating environment.

Different types of robotic applications place varying demands on language design and implementation. The programming environment is the mechanism whereby the problem specification is input to the robot programming system. Similarly, the operating environment is the vehicle for implementing the solution output by the robot programming system. This process-oriented view of robot programming motivates some of the following conclusions.

The application problem should be represented in a language providing constructs most relevant to the user. Additionally, the programming technique employed to interface with the user should be most conducive to rapid development. Taken together, these key issues form the core of a good programming environment, which in turn largely characterizes the programmability (and hence the quality) of a system.

The control of manipulators, sensors, fixtures, and tools likewise places demands on the robot programming system, particularly with respect to operating environment considerations. Since robot programming for this generation of industrial robots needs to accomplish both problem representation and control, controllers which utilize a single higher order language must address both of these problems simultaneously. This need is in contrast to computer programming languages, which tend to either expose detail (assembly level) or hide detail by data abstraction (high-level), but seldom provide both capabilities in a single language. Note that an executing robot program still implicitly requires the representation of the problem to be close at hand for the selection of alternative actions necessitated by changing conditions. The fact that 80% of a robot program is concerned with handling errors illustrates the need for an executing program to perform more than control.

The development of stronger programming environments through smarter tools, such as syntax directed editors, knowledge-based application programs, CAD/Graphics program generators and simulators, is one approach to strengthening the control-oriented manipulator level languages in the direction of problem representation. These tools serve very well to remove some burden from a user who is concerned with little else than the application problem at hand. They do not, however, actually increase the user's ability to conceptualize the problem; rather they provide powerful mechanisms for the reliable generation of programs which achieve better control over the workcell.

The development of stronger operating environments with massively parallel architectures, communication networks, and sensor integration also principally addresses the needs of control. Computational speed and precision, communication speed, type and reliability are issues addressed through improvements in the operating environment. These developments follow close on the heels of state-of-the-art computing technology.

What is still missing is a robot programming system which integrates the model of an application problem into an implementation capable of simultaneously exerting control over the environment and referencing the model. One approach to this (pioneered by NBS) uses independent communicating processes, written in a few languages, to accommodate programming at the task, manipulator, and servo levels.

Another approach which suggests itself is that of designing a higher order robot language based on the needs of users to represent their problem, and on the practical requirements imposed by robot control. It is hoped that this document has illuminated key issues in the design of current and future higher-order robot programming languages, and placed them in perspective of the industrial applications of today and tomorrow.

REFERENCES

1. Gini, G. and Gini, M., "Robot Languages in the Eighties," Robotics Assembly, Rathmill, Keith, ed., IFS (Publications) Ltd., UK, 1985.

2. Pratt, Terrence W., Programming Languages: Design and Implementation, 2nd ed., Prentice-Hall, 1984, pp 8-12.

3. Weck, M. and Niehaus, T., "Off-line Robot Programming via Standardized Interfaces," Industrial Robot, Vol. 11, No. 3, September 1984, pp 177-179.

4. D'Souza, Chris, Zuhlke, Detlef, and Blume, C., "Aspects to Achieve Standardized Programming Interfaces for Industrial Robots," Robots 7 Conference Proceedings, April 17-21, 1983.

5. Bonner, Susan and Shin, Kang G., "A Comparative Study of Robot Languages," Computer, December 1982, pp 82-96.

6. Yong, Y. F., Gleave, J. A., Green, J. L., and Bonney, M. C., "Off-line Programming of Robots," Handbook of Industrial Robotics, Nof, Shimon Y., ed., John Wiley & Sons, 1985.

7. ASEA Off-line Programming System: User's Guide, ASEA Robotics, 1986.

8. Sjolund, Paul and Donath, Max, "Robot Task Planning: Programming Using Interactive Computer Graphics," Robots 7 Conference Proceedings, April 17-21, 1983.

9. Liu, Ying-Sung, "Robot Assembly Motion Time," Robots 10 Conference Proceedings, April 20-24, 1986.

10. Govindaraj, Subbian and Doty, Keith L., "General Purpose Robot System and Task Development Facility," Robot 10 Conference Proceedings, April 20-24, 1986.

11. Volz, R. A. and Mudge, T. N., "Robots Are (Nothing More Than) Abstract Data Types," Robotics Research: The Next Five Years and Beyond Conference Proceedings, August 14-16, 1984.

12. Volz, Richard A. Woo, Anthony C., Wolter, Jan D., Mudge, Trevor N., Turney, Jerry L., and Gal, David A., "CAD, Robot Programming and Ada," NATO ASI Series, Vol. F11, Robotics and Artificial Intelligence, Springer-Verlag Berlin in Heidelberg, 1984.

13. Volz, R. A., Mudge, T. N., and Gal, D. A., "Using ADA as a Robot System Programming Language," Robots 7 Conference Proceedings, April 17-21, 1983.

14. Gruver, William A., Craig, John J., Soroka, Barry I., and Turner, Timothy L., "Evaluation of Commercially Available Robot Programming Languages," Robots 7 Conference Proceedings, April 17-21, 1983.

15. Ranky, Paul G., "Programming Industrial Robots in FMS (A Survey with Particular Reference to Off-line, High-level Robot Program Generation Using VAL, VAL-II, AML and MARTI)," Robotica, Vol. 2, Part 2, April 1984, pp 87-92.

16. Deisenroth, Michael P., "A survey of Robot Programming Languages," 1985 Annual International Industrial Engineering Conference Proceedings, December 8-11, 1985.

17. Ahmad, Shaheen, "Robot Level Programming Languages and the SRIL-90 Language," Compsac 84 Conference Proceedings, November 7-9, 1984.

18. Weck, M., Eversheim, W., Niehaus, T., Zulke, D., and Kalde, M., "Requirements for Robot Off-line Programming Shown at the Example ROBEX," Advanced Software in Robotics, Proceedings of an International Meeting, May 4-6, 1983.

19. Hollingshead, Larry L., "Elements of Industrial Robot Software," Handbook of Industrial Robotics, Nof, Shimon Y., ed., John Wiley & Sons, 1985.

20. CIMPLER(TM) Programming Manual for the CIMROC(TM) Computer Integrated Manufacturing Robot Control, GCA Corporation, Naperville, IL, July 1984.

21. KAREL Language Reference Manual, Version 1.10P, GMF Robotics Corporation, March 1986.

22. Nackman, Lee R., Lavin, Mark A., Taylor, Russell H., Dietrich, Walter C. and Grossman, David D., "AML/X: A Programming Language for Design and Manufacturing," IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1986.

23. Kempf, Karl G., "Robot Command Languages and Artificial Intelligence," Robots 6 Conference Proceedings, March 2-4, 1982.

4. Ambler, A. P., "RAPT: An Object Level Robot Programming Language," Colloquium on Languages for Industrial Robots, February 8, 1982.

25. Grossman, David D., "A Decade of Automation Research at IBM," Robots 6 Conference Proceedings, March 2-4, 1982.

26. Mogal, Joshua S., "IGRIP - A Graphics Simulation Program for Workcell Layout and Off-line Programming," Robots 10 Conference Proceedings, April 20-24, 1986.

27. Hanright, J., "Selecting Your First Arc Welding Robot - A Guide to Equipment and Features," Welding Journal, Vol. 63, No. 11, November 1984, pp 41-45.

28. Lozano-Perez, Tomas, "Robot Programming," Proceedings of the IEEE, Vol. 71 No. 7, July 1983, pp 821-841.

29. Craig, John J., Introduction to Robotics: Mechanics and Control, Addison-Wesley Publishing Company, 1986.

30. Gruver, William A., Soroka, Barry I., Craig, John J., and Turner, Timothy L., "Industrial Robot Programming Languages: A Comparative Evaluation," IEEE Transactions on Systems, Man, and Cyvbernetics, Vol. SMC-14, No. 4, July/August 1984, pp 565-570.

31. Review of the State-of-the-Art of Assembly Technologies and Programming Languages for Robotic Applications, Booze-Allen & Hamilton Inc., Arlington, VA., April 1982. Produced under subcontract for the Air Force, Contract No. F33615-80-C-5068.

32. Collins, K., Palmer, A. J., and Ratmill, K., "Development of a European Benchmark for the Comparison of Assembly Robot Programming Systems," Robotic Assembly, Rathmill, Keith, ed., IFS (Publications) Ltd., UK, 1985.

33. Saveriano, Jerry W., "An Interview with Victor Scheinman," Robotics Age, Vol. 2, No. 3, Fall 1980, pp 12-21.

34. Franklin, James W. and VanderBrug, Gordon J., "Programming Vision and Robotics Systems with RAIL(TM)," Robots 6 Conference Proceedings, March 2-4, 1982.

35. Geschke, Clifford C., "A System for Programming and Controlling Sensor-based Robot Manipulators," IEEE Transactions on pattern Analysis and Machien Intelligence, Vol. PAMI-5, No. 1, January 1983, pp 393-399.

36. Levas, Anthony and Selfridge, Mallor, "A User-Friendly High-level Robot Teaching System," International Conference on Robotics, March 13-15, 1984.

37. Shimanho, B. E., Geschke, C. C., Spalding, C. H., and Smith, P. G., "A Robot Programming System Incorporating Real-time and Supervisory Control: VAL-II," Robotic Assembly, Rathmill, Keith, ed., IFS (Publications) Ltd., UK, 1985.

38. Albus, James S., Brains, Behavior, & Robotics, BYTE Books, Subsidiary of McGraw-Hill, 1981.

39. Albus,James S., McLean, Charles R., Barbera, Anthony J., and Fitzgerald, M. L., "Hierarchical Control for Robots in an Automated Factory," Robots 7 Conference Proceedings, April 17-21, 1983.

40. Lozano-Perez, Tomas and Brooks, Rodney A., "Task-Level Manipulator Programming," Handbook of Industrial Robotics, Nof, Shimon Y. ed., John Wiley & sons, 1985.

41. Rembold, Ulrich, Dillman, R., and Levi, P., "The Role of the Computer in Robot Intelligence," Handbook of Industrial Robotics, Nof, Shimon Y., ed., John Wiley & Sons, 1985.

42. Haley, Dennis, "Intelligent Task Automation (ITA)," CAD/CAM Subcommittee 1985 Annual Report and Proceedings MTAG 85 Mini-symposium, 1985.

BIBLIOGRAPHY

The following papers represent a small sample of those published which detail specific languages available commercially or under development. For the most part, they are software or system oriented, and avoid theoretical kinematic or application discussions.

Burchardt, C. W. and Marchiando, C., "A Multi-Robot High Level Programming System for Assembly," AUTOFACT Europe Conference Proceedings, Geneva Switzerland, September 13-15, 1983.

Donato, G. and Camera, A., "A High Level Programming Language for a New Multi Arm Assembly Robot," Proceedings First International Conference on Assembly Automation, March 25-27, 1980.

Finkel, R., et al., "An Overview of AL Programming System for Automation," Proceedings Fourth International Joint Conference on Artificial Intelligence, June 3, 1975.

Gilbert, A., et al., "AR-BASIC® An Advanced and User-Friendly Programming System for Robots," Robots 8 Conference Proceedings, June 4-7, 1984.

Kirschbrown, R. H. and Dorf, R. C., "'KARMA' - A Knowledge-Based Robot Manipulation System: Determining Problem Characteristics," Robots 8 Conference Proceedings, June 4-7, 1984.

Langston, G., et al., "Robotics Software Packages in Ada," Applying Robotics in the Aerospace Industry Conference Proceedings, March 27-29, 1984.

Larson, T. M. and Coppola, A., "Flexible Language and Control System Eases Robot Programming," Electronics, Vol. 57, No. 2, June 14, 1984.

Lieberman, L. I., and Wesley, M. A., "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly," IBM Journal of Research and Development, Vol. 21, No. 4, July 1977.

Mohri, S., et al., "Assembly Robot Language and Control Software," AUTOFACT 5 Conference Proceedings, November 14-17, 1983.

Paul, R., "WAVE: A Model-Based Language for Manipulator Control," Technical paper MR76-615, Society of Manufacturing Engineers, 1976.

Paul, R., Evaluation of Manipulator Control Programming Languages," IEEE paper, 1979.

Ward, M. R. and Stoddard, K. A., "KAREL: A Programming Language for the Factory Floor," Robotics Age, September 1985.

Wood, B. O. and Fugelso, M. A., "MCL, The Manufacturing Control Language," Robots 7 Conference Proceedings, April 17-21, 1983.

A few papers contrast programming methodologies in the industrial environment. The two included here from IBM Research make an attempt to illustrate a few methodologies by considering example tasks in a few high-level languages written by IBM.

Grossman, D. D., "Robotic Software," Mechanical Engineering, August 1982.

Meyer, J., "Data Driven Automation," IBM paper, 1984.

Articles from trade journals such as these discuss off-line programming with respect to the established technologies of CAD and NC machining. Within them references are made to specific products and applications.

Gettelman, K., "The Building Blocks of Off-Line Robot Programming," Modern Machine Shop, November 1985.

Gettelman, K., "Off-Line Programming Comes to Robots," Modern Machine Shop, November 1985.

Kuvin, B. F., "Off-Line Programming Keeps Robots Working," Welding Design & Fabrication, November 1985.

These references illustrate various general approaches and techniques to using world models and CAD information for robot programming. They are somewhat conceptual, as opposed to papers which show the application of a specific language to a problem.

Duffie, N., et al., "A Sensor Based Technique for Automated Robot Programming," Journal of Manufacturing Systems, Vol. 3, No. 1, 1983.

Kent, E. W. and Albus, J. S., "Servoed World Models as Interfaces between Robot Control Systems and Sensory Data," National Bureau of Standards paper, 1982.

Nitzan, D. and Bolles, R. C., "Workshop on Intelligent Robots: Achievements and Issues," SRI International Workshop Proceedings, July 1985.

Weck, I. M., et al., "Fundamentals for the Development of a High Level Programming Language for Numerically Controlled Industrial Robots," AUTOFACT West Proceedings, November 17-20, 1980.

APPENDIX A

FIELD INTERVIEW SOURCES

Interviews with knowledgeable professionals working in robot programming languages, or allied fields, were a vital source of input to this state-of-the-art survey. The authors wish to gratefully acknowledge the contributions of the following:

Dr. Margaret A. Eastwood V.P. Products & Systems Eng.	CIMCORP 615 Enterprise St. Aurora, IL
Mr. Bertil Thorvaldsson Manager, Product Development	ASEA Robotics Inc. 16250 West Glendale Dr. New Berlin, WI 53151
Mr. Mitchell Ward Director of Software	GM Fanuc Robotics 5600 New King St. Troy, MI 48098-2696
Mr. J. E. Triggs	General Motors Artificial Intelligence Section Robotics Engineering Group Tech Center Warren, MI 48090
Dr. Robert L. Haar Staff Research Scientist	General Motors Research Laboratory Computer Science Department Warren, MI 48090
Mr. S. Shmuter, Supervisor Mr. Youssef Ali Hamidieh	Ford Motor Company Advanced Technology and Automation Development Dept. 24500 Glendale Ave. Detroit, MI 48239
Mr. Tom Helzerman, Supervisor	Ford Motor Company RAACC 15100 Mercantile Dearborn, MI 48121
Mr. M. Wudzinski	Industrial Technology Institute P.O. Box 1485 Ann Arbor, MI 48106

STATE-OF-THE-ART REVIEW
OF HIGHER ORDER LANGUAGES FOR ROBOTS

Date: October 1986

INSTRUCTIONS TO THE USER: The above publication was produced by the Manufacturing Technology Information Analysis Center (MTIAC), an Information Analysis Center administratively managed and funded by the Defense Logistics Agency (DLA). Since it is the policy of DoD and DLA that this Center be responsive to the scientific and technical information needs of the Defense community, we would appreciate it if you would complete this questionnaire and return it to us. In that way we can use your evaluation and the other information you provide us to more effectively guide this Center in meeting your needs for scientific and technical information and also to assess the value of this Center to DoD.

1. Name

2. Organization

3. Job Title

4. Field of Specialization

5. Please evaluate this publication (Check off one or more as applicable).

Information irrelevant	Difficult to use
Information relevant	Use it often
Information outdated	Hardly use it
Information timely	Overall very useful to my job
Information technically excellent	Overall not very useful to my job
Information technically satisfactory	Got my money's worth
Information technically unsatisfactory	Did not get my money's worth
Easy to use	Other_____

6. Benefits you gained by using this publication.

a. How often would you estimate that you have consulted or will consult this product?_____per day, week, month, or year (Circle one).

b. What is the average amount of time that it would take you to otherwise locate this information yourself?_____

c. What do you estimate this time would cost you?_____

d. Can you think of instances in which the information contained in this product helped to save/avoid costs on a project/task? (e.g., eliminated or shortened a test, substituted material or components) Please list these projects/tasks (e.g., Minuteman III/flight test instrumentation system) individually along with estimated costs saved/avoided.

e. Intangible benefits (please describe).

7. If you think any aspects of this publication to be inadequate, how can it be improved?

(Staple Here)

(Fold on Dotted Lines)

DEFENSE LOGISTICS AGENCY
Cameron Station
Alexandria, Virginia 22314

Official Business
Penalty for Private Use, $300

POSTAGE AND FEES PAID
DEFENSE LOGISTICS AGENCY
DOD 304

U.S. MAIL

DEFENSE TECHNICAL INFORMATION CENTER
ATTN: DTIC-DF
Cameron Station
Alexandria, VA 22314

8. Please tell us about your needs for scientific and technical information. What organized body of information would help you in doing your job? Please include such specifics as the materials, components, devices, or properties (electrical, magnetic, etc.) for which you need information. Also the format that would be most useful to you; forecast of number of years that this body of information will be required by the Defense community; forecast of total savings to you if this information was readily available.

END
DATE
FILMED
7-88
DTIC